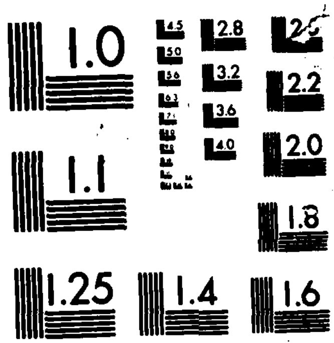
AD-A124 843 A SYNTAX-DIRECTED PROGRAMMING ENVIRONMENT FOR THE ADA PROGRAMMING LANGUAGE(U) AIR FORCE INST OF TECH WRIGHT-PATTERSON AFB OH SCHOOL OF ENGINEERING 1/2

UNCLASSIFIED S E FERGUSON DEC 82 AFIT/GCS/MA/82D-1 F/G 9/2 NL


1.0
1.1
1.25
1.4
1.6
1.8
2.0
2.2
2.8
3.2
3.6
4.0

0

AD A 124 843

A SYNTAX-DIRECTED
PROGRAMMING ENVIRONMENT
FOR THE ADA PROGRAMMING LANGUAGE

THESIS

AFIT/GCS/MA/82D-1 Scott E. Ferguson
Capt USAF

DTIC FILE COPY

DTIC
SELECTE
FEB 24 1983
S D
A

This document has been approved for public release and sale; its distribution is unlimited.

DEPARTMENT OF THE AIR FORCE
AIR UNIVERSITY (ATC)
AIR FORCE INSTITUTE OF TECHNOLOGY

Wright-Patterson Air Force Base, Ohio

83 02 023 127

AFIT/GCS/MA/82D-1

A SYNTAX-DIRECTED PROGRAMMING ENVIRONMENT FOR THE ADA PROGRAMMING LANGUAGE

THESIS

AFIT/GCS/MA/82D-1 Scott E. Ferguson
Capt USAF

DTIC
ELECTE
FEB 24 1983
D
A

Approved for public release; distribution unlimited.

AFIT/GCS/MA/82D-1

A SYNTAX-DIRECTED
PROGRAMMING ENVIRONMENT
FOR THE ADA PROGRAMMING LANGUAGE

THESIS

PRESENTED TO THE FACULTY OF THE SCHOOL OF ENGINEERING
OF THE AIR FORCE INSTITUTE OF TECHNOLOGY
IN PARTIAL FULFILLMENT OF THE
REQUIREMENTS FOR THE DEGREE OF
MASTERS OF SCIENCE IN COMPUTER ENGINEERING

Accession For

DTIC COPY INSPECTED 2

A

by

Scott E. Ferguson
Capt USAF

Graduate Computer Science

6 December 1982

Approved for public release; distribution unlimited.

PREFACE

This document describes the design and implementation of a programming support environment for the ADA programming language based on a syntax-directed editor. The original goal of the project was that of implementing only the editor. As the project progressed, the desire to see a complete environment based on the editor was overwhelming. Of tremendous credit to the editor itself is the fact that the compiler for a brief subset and the interpreter were developed in a single weekend, demonstrating further the benefits of the approach.

Much of this effort is based upon concepts and ideas from research by Bruce J. MacLennan of the Naval Postgraduate School and two of his former thesis students, William R. Shockley and Daniel P. Haddow, whose works are cited in the bibliography.

I would like to thank my advisor, Captain Roie Black, for directing me to this topic and for his stubborn refusal to accept less than my best effort on the project. Thanks are also due to my loving wife for her patience with a madman during this ordeal, Ralph (the talking computer) for all his support, and my parents, without which this project would never have been accomplished.

Scott E. Ferguson

Table of Contents

List of Figures

ABSTRACT

This document describes the design and implementation of a programming support environment for the ADA language based on a syntax-directed editor and a program tree structure. Though the prototype compiler is limited to a small subset, the full ADA language is supported by the remainder of the environment. Most of the environment is driven by a language syntax description, and is therefore capable of processing virtually any programming language. The prototype syntax-directed environment demonstrates the ability to reduce programmer idle time during development by eliminating parsing and lexical analysis in the compiler. The program tree structure also allows for the development of superior programming environment tools.

1. INTRODUCTION

ADA is the new computer programming language for embedded computer systems within the Department of Defense (DOD). Motivated by desires for increased productivity and reduced cost, ADA is one of few computer languages ever developed with the programming support environment in mind. The DOD "Stoneman" document outlines requirements for an ADA Programming Support Environment (APSE) to include many of the conventional development tools (Ref 2). The objective of this research is to present an alternative form of a software development environment designed to significantly reduce programmer idle time and increase productivity. This increase in productivity should result in lower software development costs for systems in the Air Force and the Department of Defense.

A programming environment may be thought of as providing a capability to develop programs, implying requirements for their creation, modification and evaluation. This is traditionally accomplished by a set of tools: an editor produces a text program image; a compiler generates machine code; a debugger provides a mechanism for stepping through code execution to provide runtime performance analysis and diagnostics for testing.

Modern enhancements call for an integrated collection of tools, providing a smooth and often invisible transition from one tool to the next. The desired effect is to reduce the programmer idle time between a program change and the observation of its impact upon program execution. It is this quality of the BASIC programming language environment that has made it popular with personal computer users in the microprocessor revolution, in spite of the poor reputation the language itself has obtained among many in the computer science community (Ref 4:14). Similar capabilities have been seen in other interpreted languages such as FORTH, APL and LISP. The most attractive environment currently available for a block-structured language on a microcomputer system is probably the UCSD Pascal System which attempts to address the issues of development time reduction and system integration.

In the traditional environment the program source text file is the only data structure that remains from one iteration of development to the next. Because of this, much time is being wasted during compilation by reanalyzing portions of the program which may not have been changed. The programmer, of course, usually sits idle waiting for the results of compilation. A more useful data structure to represent a program then becomes highly desirable, one which will not only record the structure of the program (and hopefully in a more efficient manner) but may also retain desirable information, avoiding lengthy and redundant reanalysis.

INTRODUCTION

Creating and maintaining this new structure requires a different sort of editor, since a text-editor is no longer appropriate. Also, since the program's structure is now so readily accessable, the editor might be built to ensure that this structure is correct as it is being entered. Much of this effort can be accomplished using time which, in single-user or highly interactive environment, is often wasted by the computer in idle loops during program entry waiting for another typed character. Editors of this type are called syntax-directed editors. The use of a syntax-directed editor frees the compiler from the time-consuming task of determining if a program's structure is correct.

The program data structure used by most syntax-directed editors is that of a program tree. This tree becomes a major focal point where analysis information may be retained and exchanged between tools within the environment. Just what a program tree looks like, how it is created and used, is a major topic of this report.

The goal of this research effort is to design and implement a programming language environment based upon a syntax-directed editor to support the analysis of such an environment and provide a foundation for future efforts. The chosen target language is ADA, although a small subset was chosen as a more realistic goal within time constraints and for the sake of demonstration. The system described was implemented on a Z80-based microcomputer system using the

CP/M operating system. Software was developed in the C programming language, chosen for its suitability to the problem, compactness of code for a small machine and high-level language portability.

Chapter 2 of this report provides some background on the process of defining the structure of a language. How such a language structure definition is used by the syntax-directed editor during the synthesis of program trees is the topic of Chapter 3. Next, Chapter 4 addresses the problems involved with generating a conventional text display from a program tree. Chapters 5 through 7 describe the functions of the compiler, interpreter/debugger and program lister of the prototype syntax-directed environment. The implementation of these programming tools, including the editor, is discusssed briefly in chapter 8. Chapter 9 outlines some more advanced concepts for improvement of the system. Finally, chapter 10 presents the conclusions and recommendations resulting from the research effort.

2. LANGUAGE SYNTAX SPECIFICATION

A language's syntax (or grammar) precisely defines the rules which govern the structure of programs written in the language. Several means have been developed to specify a language's syntax. One such means was proposed by Niklaus Wirth and is a language in itself called Extended Backus-Naur Form, or Extended BNF (Ref 14). Using Extended BNF, the language designer describes the appearance of each language construct in detail. (The language semantics, the operational effect of a program's statements, are not addressed by a syntax description.)

The ADA Reference Manual, as an example, uses a form of Extended BNF to describe the ADA programming language (Ref 3). The ADA assignment statement is defined in a syntax rule as:

```
assignment_statement ::=
    variable_name := expression;
```

The name of the language construct being defined appears first. The symbol "::=" is read "is defined as", and immediately precedes the definition. The definition specifies the order of appearance of elements in the language. The tokens ":=" and ";", called terminals, appear in the program as they do in the definition. "variable_name" and "expression" are non-terminals representing other language constructs with their own definitions presented elsewhere in the syntax description.

In an ADA program, an assignment statement might appear as in figure 2-1.

```
NEWX := OLDX*SCALEX;
```

Figure 2-1. An ADA assignment statement.

"NEWX" and "OLDX*SCALEX" are said to be produced from the definitions of variable_name and expression, respectively. For this reason, syntax rules are often called productions.

As a compiler analyzes a source program, it breaks it down into its language construct components through a process called parsing. This is often accompanied by the construction of a parse tree for the program. A parse tree records the hierarchical structure of a program. Each terminal and non-terminal used in a production definition is organized below the non-terminal being defined. A simple parse tree fragment for the assignment statement above is shown in figure 2-2.

The definitions for variable_name and expression are used to complete the structure of the parse tree below them. The program tree data structure used by the syntax-directed editor is very similar to the parse tree. The techniques for constructing a program tree using a syntax description are the topic of the next chapter.

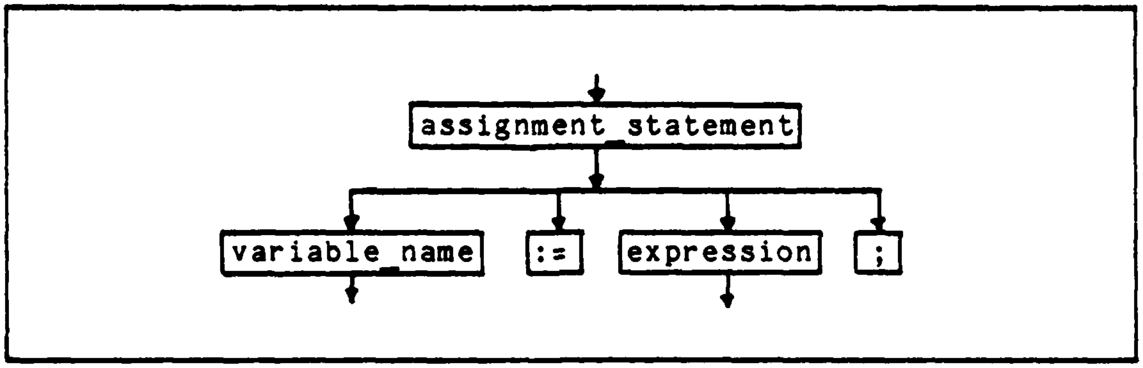


Figure 2-2. Parse tree for ADA assignment statement.

Syntax description languages, such as Extended BNF, have been used in the development of compilers. Compilers are normally concerned with determining how a particular program was constructed from the rules of the language syntax. The syntax-directed editor, however, is concerned with the opposite task of constructing a program using syntax rules. The syntax-directed editor requires additional formatting information to construct the textual image of the program from the program tree. Also since the syntax rules directly control program tree construction, they should be kept simple and efficient. For these reasons, Wirth's Extended BNF, was augmented to create META, a syntax definition language used to describe a programming language's syntax and its display format. META's form and use will be presented during the discussion of program tree synthesis in chapter 3. A syntax definition of META written in META is provided in appendix I. META was greatly influenced by the R-ARGOT syntax notation of another editor (Ref 11).

LANGUAGE SYNTAX SPECIFICATION

A META description for the full ADA programming language, derived from the 1980 ADA reference manual (Ref 3), is given as appendix II and provides examples used throughout this document.

3. PROGRAM TREE SYNTHESIS

This chapter describes the synthesis of a program tree in the ADA programming language. The ADA syntax description features and their effect upon program tree synthesis and display are examined.

3.1 Creating an ADA Program Tree

The ADA syntax description is presented as a sequence of production rules. The first production rule defines the language goal symbol. The definition of this goal symbol describes the structure of the entire language in terms of other language terminals and non-terminals. The goal symbol is therefore used to form the root of a new program tree. Since the ADA goal symbol is a compilation, the creation of a new ADA program tree begins with a compilation node as in figure 3-1.

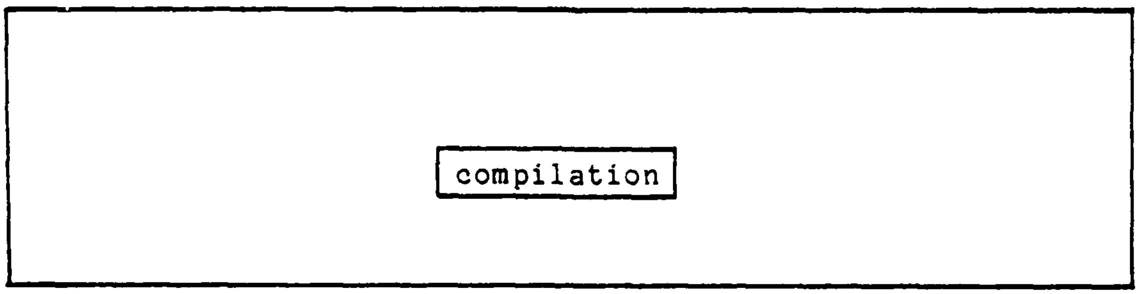


Figure 3-1. ADA program tree root.

The production definition of compilation is:

```
compilation =
     compilation_unit
     { @ compilation_unit } ;
```

The element being defined is named first, followed by an equals sign ("="), the definition, and terminated with a semicolon (";"). The "@" symbol, used for display control, will be discussed later.

This is an example of a concatenation type of production. The application of this production to the compilation node results in the creation of a new child node below the compilation node for each element in the definition. The braces ("{" and "}") surrounding the second element indicate that it may appear zero or more times. Such an element is called a repeater and will be conditionally added to the tree to allow the user the option of establishing it as part of the program, or removing it. Until established, these tree nodes are ignored by environment tools such as the compiler as if they did not exist. The user may also insert additional nodes of this type into the tree as desired.

The compilation production transforms the tree of figure 3-1 to that of figure 3-2.

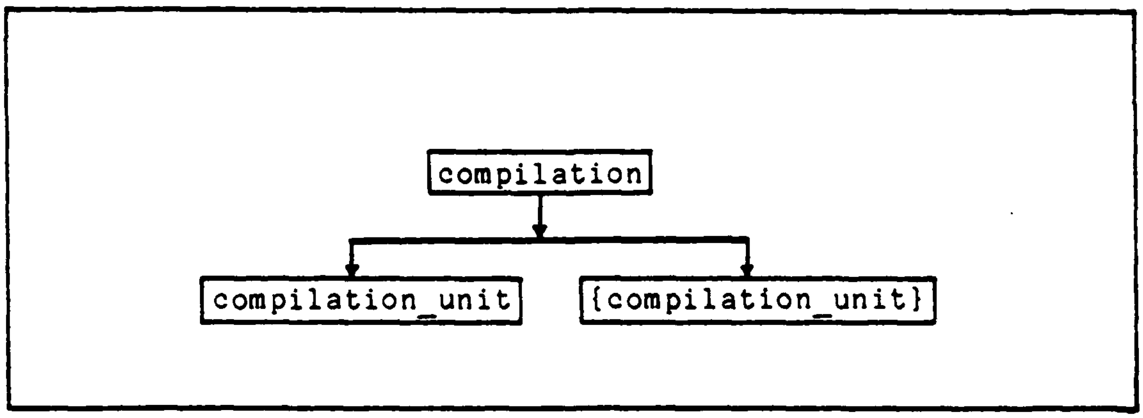


Figure 3-2. Program tree with compilation_units.

```
<compilation_unit>
{<compilation_unit>}
```

Figure 3-3. Display of program tree with compilation_units.

The program tree's display image at this point is shown in figure 3-3. The image is generated from the program tree <u>frontier</u> (those tree nodes on the border of the tree which have no children) by examining the rules and display control information. The <u>interior</u> nodes (all others) are not directly displayed.

The <u>newline</u> display format control ("@") in the definition causes the second compilation_unit to be displayed on a separate line. The angle brackets ("<" and ">") are added to the display to avoid confusing the non-terminal names with other elements of the program that may appear later. This choice is arbitrary. Non-terminals might just as easily be distinguished by some other means such as highlighting, color change or alternate character font where such capabilities exist. Note also the use of braces in the display to reflect the conditional nature of the second compilation unit.

The first compilation_unit node may now be examined for potential synthesis operations. A compilation_unit is defined as:

```
compilation_unit = <
    proc_body
    func_body
    pack_body
    proc_decl
    func_decl
    pack_decl
    with_use_clause
    subunit
    pragma >;
```

The angle brackets surrounding the definition designate this production as an alternation. Each alternative must be a single element. This production has nine alternatives, one of which is to be selected by the user for synthesis into the tree. The programmer might decide at this point if this portion of the program should be, for instance, a procedure body. The list of alternatives is displayed for the user to aid the selection process. The implemented prototype editor requires the user to select an alternative by typing the name of the alternative, with command completion to help speed the selection. The use of a pick device (such as a light pen or a mouse) would be quite efficient where such capabilities exist.

Assuming that a proc_body was selected, a new child is created below the first compilation_unit of figure 3-2 to record the selection and transform the program tree to that shown in figure 3-4.

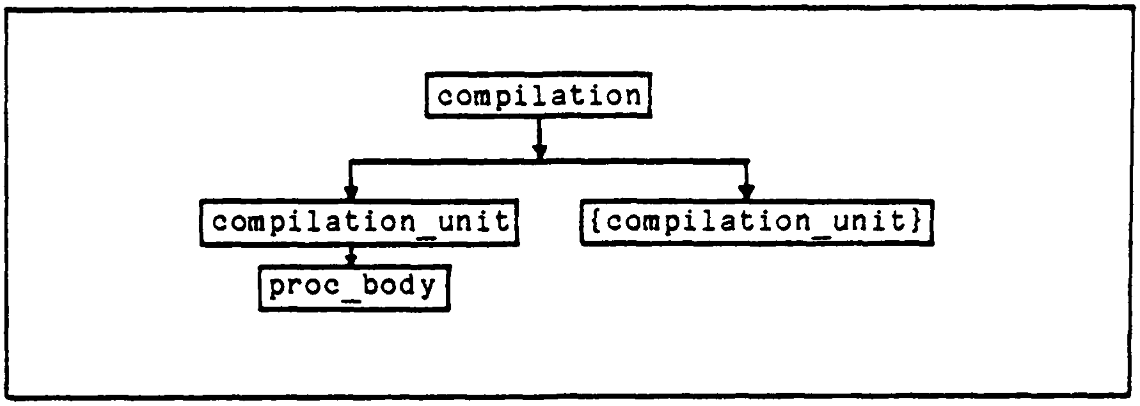


Figure 3-4. Program tree with proc_body alternative.

The next synthesis action is directed by the production definition for proc_body:

```
proc_body =
     proc_spec ^ "is"
          { + decl }
          { + rep_spec ! }
          { + program_component }
     @ "begin"
          + seq_of_stmts
     [ @ exceptions ! ]
     @ "end" [ ^ identifier ] ";" ;
```

As is to be expected, this production lays out a template of what a procedure body is to look like. This is another concatenation definition causing the creation of a tree node below the proc_body node for each element in the definition. The brackets ("[" and "]") surrounding an element indicate that it may appear once or not at all. Such an element is called an <u>option</u> and, like a repeater, will be synthesized into the tree in a conditional manner to allow the user the option of establishing it as part of the program, or removing it.

Options and repeaters are conditional elements not required to create a valid program in the language. In some instances the synthesis of certain conditional elements may only rarely be desired and it may be preferable to require user action to insert them rather than to remove them, thus reducing the number of editing keystrokes for the usual case. The hide indicator ("!") may be used to mark conditional elements (such as exceptions or rep_spec in the example) so they will not be automatically constructed in the tree. The user must manually insert such an element to get one in the program and on the display.

The quoted strings are terminal language elements typically used to represent reserved words and delimiters in the language. They stand for themselves and thus need not be defined. A terminal string included unconditionally in a concatenation list represents an invariant field which does not record a decision in the tree synthesis process and offers no potential for growth of the tree below it. Other research has confirmed the intuitive suggestion that such strings need not occupy space in the tree (Ref 11:61). In an effort to reduce the size of the program tree, therefore, such strings are not synthesized in the tree.

Applying the proc_body production to the proc_body node transforms the program tree of figure 3-4 to that of figure 3-5.

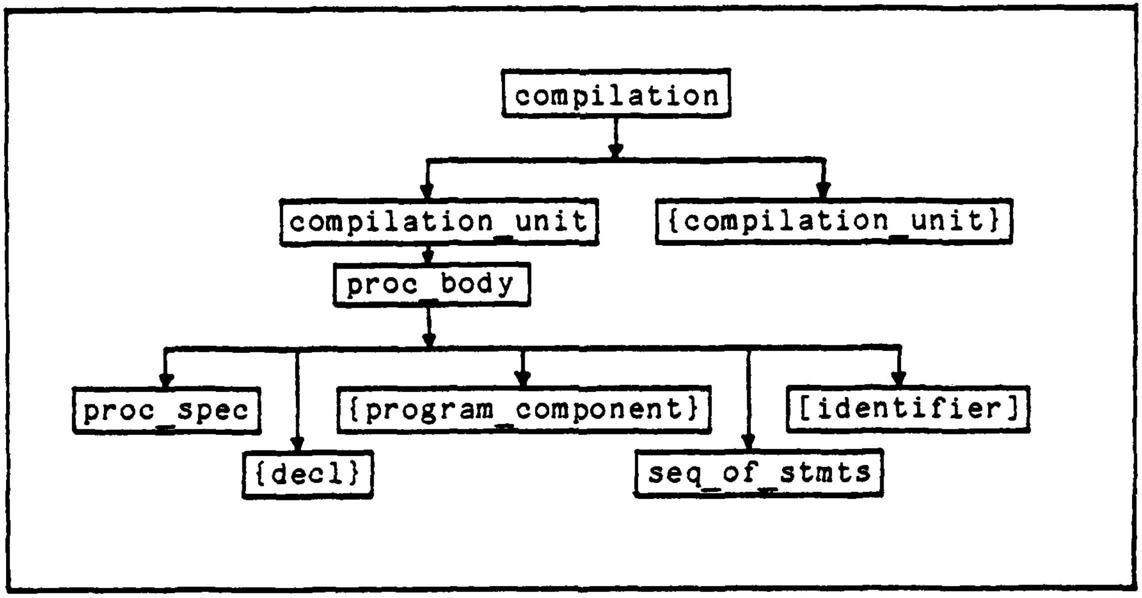


Figure 3-5. Program tree after applying proc_body production.

Note that the conditional elements, rep_spec and exceptions, were not automatically synthesized in the tree because of their hide markers. If desired, they must be inserted by the user. The terminal strings in the production are also not present in the tree.

```
<proc_spec> is
     {<decl>}
     {<program_component>}
begin
     <seq_of_stmts>
end [<identifier>];
{<compilation_unit>}
```

Figure 3-6. Display after applying proc_body production.

The program tree's current display image is shown in figure 3-6. The space marks ("^") in the definition cause a space to be inserted in the program tree image. An indentation mark ("+") preceeding an element indicates that the image for that element (represented by its subtree) is to be indented in the display. The optional identifier has been distinguished by brackets. The presence of strings in the display image of the program tree are reconstructed from the production definition.

The enforcement of a standard layout format by syntax-directed editors is common. The specification of format controls in the syntax definition, as provided here, allows some measure of local control for individual tastes.

Note that at this point the only user action has been to select a procedure body for the program. All the reserved words and semicolons have been properly placed by the editor. In addition, the editor has included information about the types and locations of language constructs that remain to be supplied by the user to complete the program correctly.

At this point only two nodes on the frontier of the tree (proc_spec and seq_of_stmts) are unconditional. The remainder are conditional nodes (options and repeaters) that will require some user action to establish them as areas of the program to be expanded upon. If the definition of either remaining unconditional node is an alternation, it too will require user action in the form of a selection before synthesis may continue using that node. The definition of proc_spec:

```
proc_spec =
    "procedure" ^ identifier [ formal_part ] ;
```

is, however, a concatenation definition. No user action is required to apply the proc_spec production to the tree. The proc_spec node is not an option or repeater, so it must be present to satisfy the syntax of the language. Therefore, whenever a new unconditional node, such as this, with a concatenation definition is synthesized into the tree, its definition production may be automatically applied. In this manner, the proc_spec production should be automatically applied when the proc_body production is applied to produce the subtree of figure 3-7.

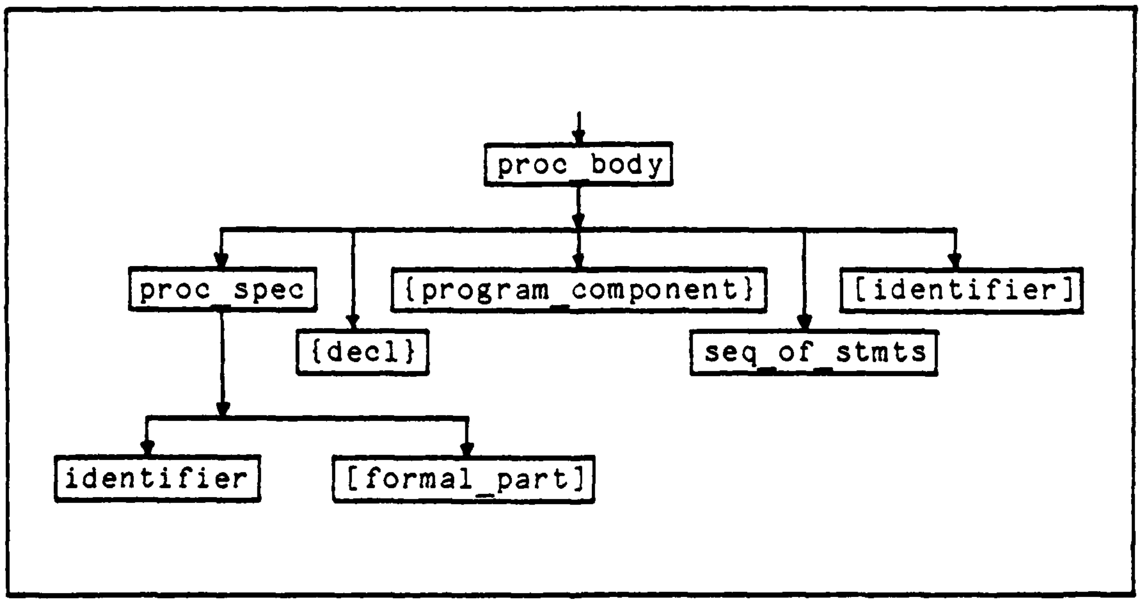


Figure 3-7. Proc_body subtree after automatic application of proc_spec production.

This application of concatenation productions upon unconditional nodes is automatic whenever such a node is added to the tree and continues until no such nodes remain. Likewise, the production definition for identifier:

```
identifier =
     'AZ¦az' { '09¦AZ¦_¦az' } ;
```

is also applied to identifier node under proc_spec.

The first element of the definition, 'AZ¦az', is an example of the <u>set</u> construct which represents a compact means of expressing an alternation consisting of single-character strings. The set's alternatives may be single characters or pairs of characters which specify an inclusive range in the ASCII character set. The set 'AZ¦az' is syntactically equivalent to the alternation definition:

```
letter = <
     "A" "B" "C" "D" "E" "F" "G" "H" "I" "J" "K"
     "L" "M" "N" "O" "P" "Q" "R" "S" "T" "U" "V"
     "W" "X" "Y" "Z" "a" "b" "c" "d" "e" "f" "g"
     "h" "i" "j" "k" "l" "m" "n" "o" "p" "q" "r"
     "s" "t" "u" "v" "w" "x" "y" "z" >;
```

Besides being obviously easier to use, the set saves a program tree node by storing the user-selected character as an attribute of the node. The letter production definition, being an alternation definition, would require the selected letter to be made a child of the letter node.

The identifier production applied to the identifier node produces the subtree and display of figure 3-8.

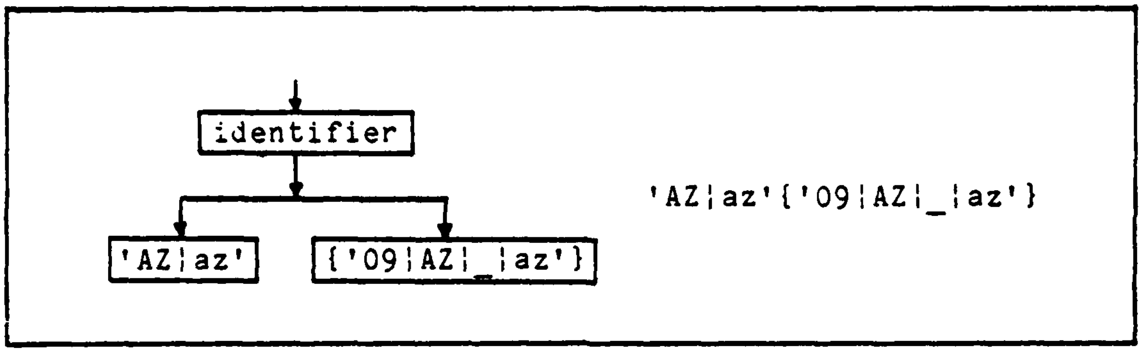


Figure 3-8. Identifier subtree and display.

The user selects (by typing) one character from the set's valid range. This character then replaces the set name in the display of the set node. Typing an 'X' for the 'AZ¦az' set node, for example, would transforms the subtree of figure 3-8 to that of figure 3-9.

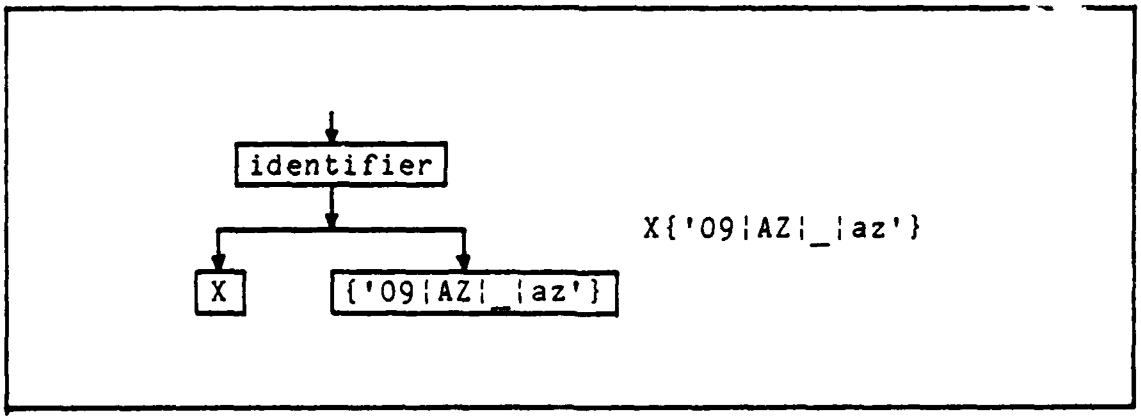


Figure 3-9. Identifier with first character.

3.2 Conditional Node Establishment

Conditional leaf nodes, such as the second set node in the identifier above, require user interaction to establish them into the tree. If the node is defined as a concatenation or a string, typing the first character of its displayed name will establish the node. If the node is defined as an alternation or a set, this interaction is implied by selecting an alternative or typing a character, respectively, as previously described.

When established, the node's surrounding brackets or braces are no longer displayed. If the conditional being established is a repeater, a new, unestablished repeater of the same type is automatically inserted after the node in anticipation of the user's desire to later establish another such node in the tree. In this manner, a stream of characters may be input for an identifier or additional statements supplied in a sequence of statements without having to manually insert each subsequent one.

Consider the identifier node from figure 3-9. Typing a 'Y' establishes the second set node and becomes the second character of the identifier. A copy of the set repeater node is automatically inserted into the tree for entering the next character as shown in figure 3-10.

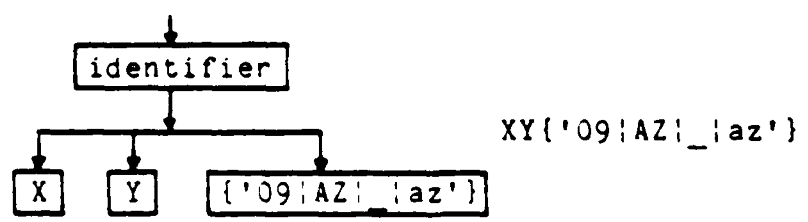


Figure 3-10. Identifier with second character.

The unestablished repeater set node may be manually deleted when no more characters are to be added to the identifier.

3.3 Insertion and Deletion

The program tree may be modified during its synthesis with insert and delete operations. The editing <u>focus</u> is the current program tree node of interest to the user. The entire subtree below the focus designates the entity being considered by the editor for manipulation by the next editing command. The delete operation causes the focus subtree to be deleted from the program. If the focus is at an option or repeater node, it represents an unnecessary (i.e. conditional) node and may be simply deleted. Otherwise the node is required by the syntactic definition of its parent. In this case, the node is retained but all those which have grown from it are deleted to wipe the subtree clean and allow it to be rebuilt. To completely delete such a node requires that its parent be deleted.

If the procedure body, previously shown in figure 3-7, was to have no formal parameters, the focus would be moved to the "formal_part" node and the delete operation would be invoked, resulting in the subtree of figure 3-11.

The insert operation may be used to add an optional or repeater node to the tree. The "formal_part" of the procedure may be restored to its previous position by moving the focus to the "identifier" node and performing an insert

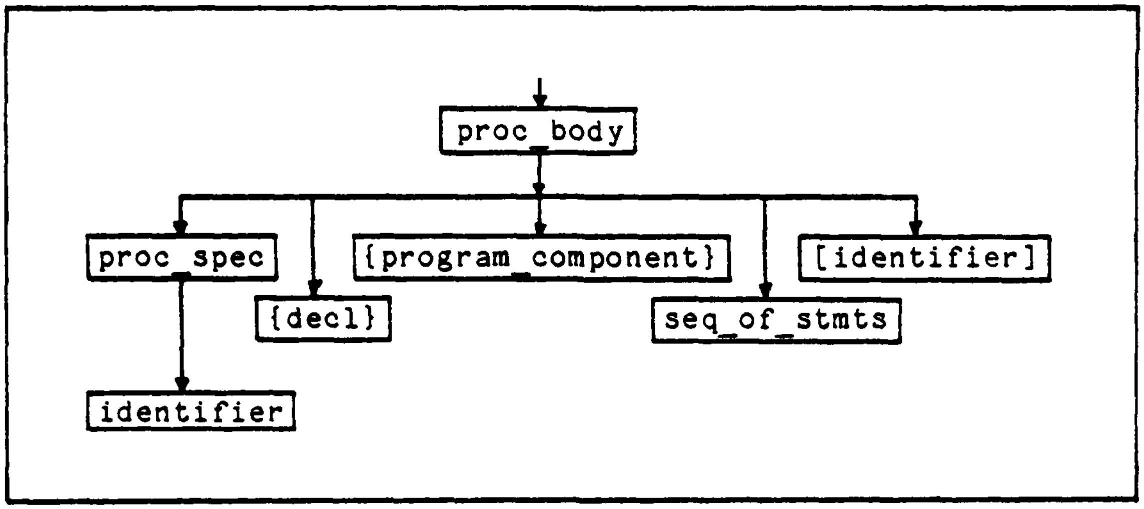


Figure 3-11. Proc_body subtree after deleting the formal_part.

operation. The type of node to be inserted is determined from the production definition of the parent (in this case the "proc_spec" node) and the position of the focus with respect to this definition. An option or repeater must be capable of being inserted at that point. Only one occurance of an optional node may be inserted into the tree at one time, while a repeater may be inserted as many times as desired. A similar operation is required to insert a node in the tree to the left of the focus.

3.4 Cut and Paste Editing

"Cut and paste" operations can be easily applied to the program tree structure. An entire subtree (such as that of the proc_body above) can be clipped or copied from the tree and reattached in place of any proc_body node in the tree. This is a good example of how the syntax-directed editor treats the program as program units rather than raw text.

3.5 Language Subsetting

The tree synthesis process, as just described, is guided directly by the syntax description of the language. Elements removed from the syntax description can no longer be synthesized by the editor. Subsets of a language can be created through a controlled removal of unwanted or unneeded language features. Subsets may be useful at some installations which, for example, wish to restrict the use of certain language features possessing potentially dangerous power. In the academic environment instructors may want to disallow the use of certain programming constructs until they are properly introduced. The dreaded "goto" may also be a desirable target for omission from a language.

Programs generated from the subset should be compilable by the compiler for the entire language. This requirement is assured, in a syntactic sense, if any program produced from the subset grammar is producible from the original grammar. This property determines the type of syntactic elements, discussed below, that may be removed to produce a subset grammar.

Conditional elements (options and repeaters) represent syntactic units that may be omitted while still producing a syntactically correct program. It follows that a conditional element of the language grammar may be removed to produce a new grammar, and that any program produced from the new grammar is producible from the original. Individual alternatives within an alternation are likewise candidates for removal since syntactically valid programs can be created without chosing a particular alternative.

Alternative and conditional elements in a META language description may be marked with a subset index indicator which is a dollar-sign ("$") followed by a series of digits ("0" to "7"). Each digit indicates a subset within which that element is "turned off". An element marked with subset index $03, for example, would be disallowed when using subsets 0 or 3 of the language or both. In this manner up to 256 grammar subsets may be supported by a single META description.

The META definition for ADA presented in appendix II has been reduced to a more manageable subset for the implementation of a full prototype syntax-directed programming language environment. The ADA0 definition given in appendix III is the $0 subset of the full ADA definition. As an example of how subsetting works, the META definition for a procedure call in ADA is:

```
proc_call =
     name [ actual_param_part $0 ] ";" ;
```

The "$0" subset removes parameters from procedures resulting in the effective definition for ADA0 being:

```
proc_call =
     name ";" ;
```

4. DISPLAYING PROGRAM TREES

Because the program tree is a structure so different from a text file, the syntax-directed editor is faced with additional problems of how to display the program on the screen and how to provide feedback to the user as to movement of the focus within the tree. Certain advantages also arise from the solutions to these problems as they are discussed below.

4.1 The Extended Cursor

When editing programs using a text-based screen-oriented editor, movement within the text is usually accomplished with cursor control actions such as up, down, left and right. Each movement places the cursor in a cell where characters may be added or changed. The process works well for text-editing since a character cell represents both the unit of change and of movement.

The program tree editor wants to work with program components, however, not characters. This makes conventional screen-relative cursor movement inappropriate and makes the operation of mapping screen coordinates to the tree structure too complex. Also, many elements in the display of a program tree are not individually modifiable, such as characters within reserved words. The user might only be frustrated by being allowed to move to invariable parts of the display.

Making the tree node the unit of movement for the editor is accomplished by supplying a variety of commands to move the focus from one node to another. The entire subtree rooted at the focus designates the entity being considered by the editor for manipulation by the next editing command. The cursor should, therefore, designate the image which represents the entire focus. The point cursor of display terminals, however, actually points to only one character on the screen. This results in undesirable ambiguity as in our previous example of the identifier. If the focus is currently on the identifier node, the point cursor lights on the leftmost character of its image and the display looks like figure 4-1.

```
'AZ¦az'{'09¦AZ¦_¦az'}
^
```

Figure 4-1. Point cursor display.

But if the focus is moved down to the set 'AZ¦az', the cursor remains fixed since it already designates the leftmost character of the set display image.

The solution to this problem is the creation of an <u>extended</u> <u>cursor</u> by highlighting the display image of the focus subtree to clearly indicate the portion of the program being examined. Highlighting requires the terminal capability to display text with reverse video, color change or altered intensity. The extended cursor, appearing as an enveloping box in figure 4-2, shows the focus at the identifier node and after moving it down to the set node of 'AZ¦az'.

```
'AZ¦az'{'09¦AZ¦_¦az'}

'AZ¦az' {'09¦AZ¦_¦az'}
```

Figure 4-2. Displays using the extended cursor.

Still, implementors of other syntax-directed editors have chosen to retain the point cursor and other text-based editor characteristics, apparently in an attempt to appear more conventional (Ref 12:15).

Focus movement to adjacent nodes such as parent, child or sibling forms the primary type of movement within the program tree. It is common for keys such as "left" and "right" as found on most terminals to be interpreted as movements to siblings left and right of the focus in the program tree. Likewise, "up" and "down" translate to movements to the parent and child of the focus. Since the display format of the tree is dependent upon format controls in the syntax description, it is quite possible that siblings of the same parent will not lie on the same display line. This results in focus movement on the screen which may be inconsistent with the usual action with a text editor. Figure 4-3 shows the procedure displayed with the focus on the "decl" node. Using the "right" command moves the focus down on the screen to its right sibling (the program_component node), as shown in figure 4-4.

```
procedure <identifier>[formal_part] is
    {<decl>}
    {<program_component>}
begin
    <seq_of_stmts>
end [<identifier>];
{<compilation_unit>}
```

Figure 4-3. Focus at decl node.

```
procedure <identifier>[formal_part] is
    {<decl>}
    {<program_component>}
begin
    <seq_of_stmts>
end [<identifier>];
{<compilation_unit>}
```

Figure 4-4. Focus at program_component node.

4.2 Automatic Display Justification

A terminal display screen imposes severe constraints as to how much of a program tree may be displayed at one time. Determining what portion of the tree to display is a problem approached in many ways by different implementors of syntax-directed editors. One such implementation offers a very simple approach: allow the user to specify the subtree to be displayed (Ref 10:8). The subtree displayed must be cropped to fit the viewing surface. A depth limit is also enforced, suppressing the display of portions of the tree which are beyond a specified depth below the subtree root, to make additional room on the screen. The selection of the subtree to display and the depth limit are specified by the user. Two enhancements to this concept have been adopted for display image generation in the prototype syntax-directed environment: automatic display justification and modular subtree elision.

Constant repositioning of the subtree window by the user is considered undesirable. An assumption that the user is interested only in subtrees which include the focus allows the editor to automatically select the subtree to be displayed. The program subtree chosen for display on the terminal screen, then, is that which contains the focus but whose size does not exceed that of the screen. The focus, of course, may contain a subtree whose image is arbitrarily large. When even the focus is too large to display on the screen, it is clipped to show as much as possible.

4.3 Modular Elision

Using the depth limit approach creates the situation where the focus node will not be displayed if it is below the specified depth. Proposed here as an alternative to depth limits is a different means of suppressing detail from the display, called elision.

The display of any subtree in the prototype environment may be suppressed upon command of the user. The image of the "elided" subtree is replaced by an arbitrary string to mark the elided material. This allows suppression of undesirable detail from the display under user control.

When the focus is at or within an elided subtree, the automatic display justification mechanism will consider no larger subtree for display within the window. This results in the somewhat opposite effect of suppressing the display of all but the elided subtree.

In essence, the subtree elision concept may be thought of as specifying levels of modularity to the program tree display process. By marking a subtree for elision, the user is interpreted as saying that it is to be displayed as a modular unit. That when the focus is outside of the subtree its contents are of no specific interest, and that when the focus is inside the subtree only its contents are of specific interest. As an example, consider a program consisting of two very simple nested procedures where the focus is at the inner proc_body as in figure 4-5.

```
procedure outer is
    procedure inner is
    begin
        <seq_of_stmts>
    end inner;
begin
    <seq_of_stmts>
end outer;
```

Figure 4-5. Two nested procedures.

Eliding the inner procedure causes the display to isolate upon it as in figure 4-6.

```
procedure inner is
begin
    <seq_of_stmts>
end inner;
```

Figure 4-6. Isolation of inner procedure.

So long as the focus remains inside the inner procedure, the display shows only that procedure, removing details from the outer world. In figure 4.7, moving the focus out of the inner procedure reveals how the inner procedure is suppressed from the display.

```
procedure [outer] is

    ++++
begin
    <seq_of_stmts>
end outer;
```

Figure 4.7 Outer procedure with suppressed inner procedure.

The user remains aware that something is there, but is unconcerned with the details of its construction. It is suggested that the modular elision capability graphically supports sound programming techniques such as modular programming and stepwise refinement.

Program tree synthesis and display, the topic of the previous two chapters, are the primary functions of the syntax-directed editor. The following chapters briefly describe the other program development tools of the syntax-directed environment.

5. THE COMPILER

The compiler function of the prototype syntax-directed programming language environment is the only function which is language dependent. No means have been provided as yet to specify semantic action in the META syntax description to support program tree analysis and code generation. The compiler must be coded by hand to perform these tasks. The prototype compiler for the ADA0 subset merely walks the tree building a symbol table and generating code for a pseudo-machine.

5.1 Program Tree Walk

The structure and assured syntactic correctness of the program tree completely eliminate the need for a parser in the conventional sense. Walking the tree provides access to the syntactic elements of the program. The structure of the compiler code to walk the tree directly models the hierarchical structure of the language syntax description.

Each non-terminal in the syntax description maps to a subroutine whose argument is the root node of a subtree of the non-terminal's type. The routines's purpose is to process and validate the structure of the subtree. Thus a call to a "compilation" subroutine with an argument of the program tree root (compilation) node initiates the entire process of compiling an ADA program. Each subroutine called processes its own children in turn by calling other subroutines for non-terminals or observing strings and sets for their semantic value. As an example, the tree structure for the ADA if_stmt, syntactically defined as:

```
if_stmt =
    "if" ^ expression ^ "then"
          + seq_of_stmts
    { @ elsif_part }
    [ @ else_part ]
    @ "end" ^ "if" ";" ;
```

is processed by a subroutine like the one in figure 5-1.

```
procedure IF_STMT(node : tree_node);
     var child : tree_node;
begin
     child := first_child(node);
     EXPRESSION(child);
     child := right_sibling(child);
     SEQ_OF_STMTS(child);
     child := right_sibling(child);
     while (node_type(child) = "elsif_part")
          begin
               ELSIF_PART(child);
               child := right_sibling(child);
          end;
     if (node_type(child) = "else_part")
          ELSE_PART(child);
end;
```

Figure 5-1. Subroutine to process the ADA if statement.

Processing for the concatenation definition of the if_stmt maps to a series of statements to analyze each child node in turn. The terminal strings are of no concern here since they are not synthesized in the tree. The expression and seq_of_stmts are non-terminals processed by their own subroutines. The elsif_part repeater nodes are processed in a while-loop for as many such nodes as may be in the if statement. The optional else_part is processed in an if statement predicated upon whether or not one such node is present. Any unestablished repeaters or options are ignored.

An alternation definition is processed with a case statement (or switch) based on the type of its single child. This child, which recorded the alternative chosen, is used to determine the type of program construct which is formed in the tree below it. If it has no child, an incomplete program fragment is detected as an error.

Since the code to perform the tree walk as just described maps so directly to the syntax description, it could be automatically generated from the syntax description file in a straightforward manner. Techniques for this have been previously developed to automatically generate parsers for compilers (Ref 9). The tree walk code so developed would form a shell to be augmented with the necessary analysis and code generation routines for the language.

An additional benefit of the tree structure is that the program need not be accessed in a strict linear fashion, as has been described. The entire tree is available for analysis throughout the compilation process.

5.2 Symbol Table

The symbol table used in the prototype compiler references identifiers with a pointer to an identifier node in the tree. The individual characters are distributed as children of the identifier node according to the syntax definition for identifier. A subsequent identifier encountered in the program tree may be compared to those in the symbol table by pattern matching identifier subtrees. This leaves the storage for symbols in the trees.

5.3 Code Generation

Code generated by the prototype compiler is for a pseudo-machine similar to that used by Wirth in his Pascal subset compiler, PL/0 (Ref 13). Code is generated during the tree walk. Each instruction is tagged with a pointer to the tree node which is to be considered responsible for its generation. This pointer is used by the interpreter, discussed later.

5.4 Error Handling

The syntax-directed editor insures only that programs are free from syntactic errors as specified by the language grammar. Semantic errors, such as encountering an identifier that is either undeclared or of the wrong type, must be handled by the compiler.

The error recovery function is nearly eliminated in the compiler. Since nearly all errors are semantic in nature, recovery in the prototype involves possibly patching up code generation and resuming compilation at some arbitrary point later in the tree. An undeclared variable encountered in an expression, for example, causes an error report with an instruction generated to load a constant 0 rather than the variable value. The remainder of the expression may processed normally.

The handling of errors will at some point require the user to make program modifications. The compiler aids this process by marking faulted nodes in the program tree so the editor can be automatically directed to them. The user may repair them quickly and return to compilation.

6. THE DYNAMIC INTERPRETER/DEBUGGER

The interpreter for the prototype syntax-directed environment is perhaps the most exciting component of the environment. The interpreter provides a visual form of trace for program execution. Though only partially implemented with very minimal debugging facilities, it is representative of the kind of programming development tools that can be developed to draw upon the potential power of the program tree structure.

The interpreter follows code generation by the compiler. Each generated instruction is tagged with a pointer to the program tree node considered responsible for its generation. This node becomes the highlighted focus for the program tree display as each instruction is executed to show the programmer where execution is taking place. The focus moves across the program image as instructions are executed to dynamically trace out program execution.

The topmost elements of the runtime stack are displayed after each instruction is executed, along with the next instruction to be executed. More advanced debugging facilities, such as those described in previous research for use in the ADA environment (Ref 6), should eventually be incorporated into the interpreter/debugger.

7. THE PROGRAM LISTER

The program lister produces a text format source file from the program tree for use in generating hardcopy listings or for transfer to a text-based environment. The process is virtually identical to that of generating a program tree display image, but without a line limit restriction. The ability to pretty-print a program, a topic of considerable interest in the literature, is inherent in the structure and display of the program tree.

The resulting text file is intended to represent a program which could be presented to a conventional compiler. The lister, therefore, omits unestablished conditional nodes (since these are ignored by the compiler) and does not suppress the display of elided subtrees. Options might later be added to alter these defaults and to selectively list portions of the program, perhaps using the elision facility. The lister is envisioned as a tool which should also eventually generate cross-referencing information for programs.

8. THE PROTOTYPE IMPLEMENTATION

The prototype syntax-directed editing environment was implementated on a Heathkit H89 microcomputer system with a four megahertz Z80 processor, 64 kilobytes of memory and 600 kilobytes of floppy-disc storage. The environment has also been successfully demonstrated on an S-100 based microcomputer system with a ten megabyte Winchester disc. Software was developed for the CP/M operating system using the C programming language. A brief discussion of the implementation highlights follows.

8.1 System Organization

The environment is organized with the program tree storage at the center, as depicted in figure 8-1.

The syntax directed editor is the primary user interface of the environment. Control is transferred to the compiler, interpreter or lister upon command from the editor. The META preprocessor condenses a textual META syntax definition into a form more easily accessed by environment tools. The terminal configuration program records pertinent terminal characteristics in a terminal description. Interfaces within the environment are described briefly following a discussion of the global software packages providing access to the program tree structure and syntax description structure and routines for program display generation.

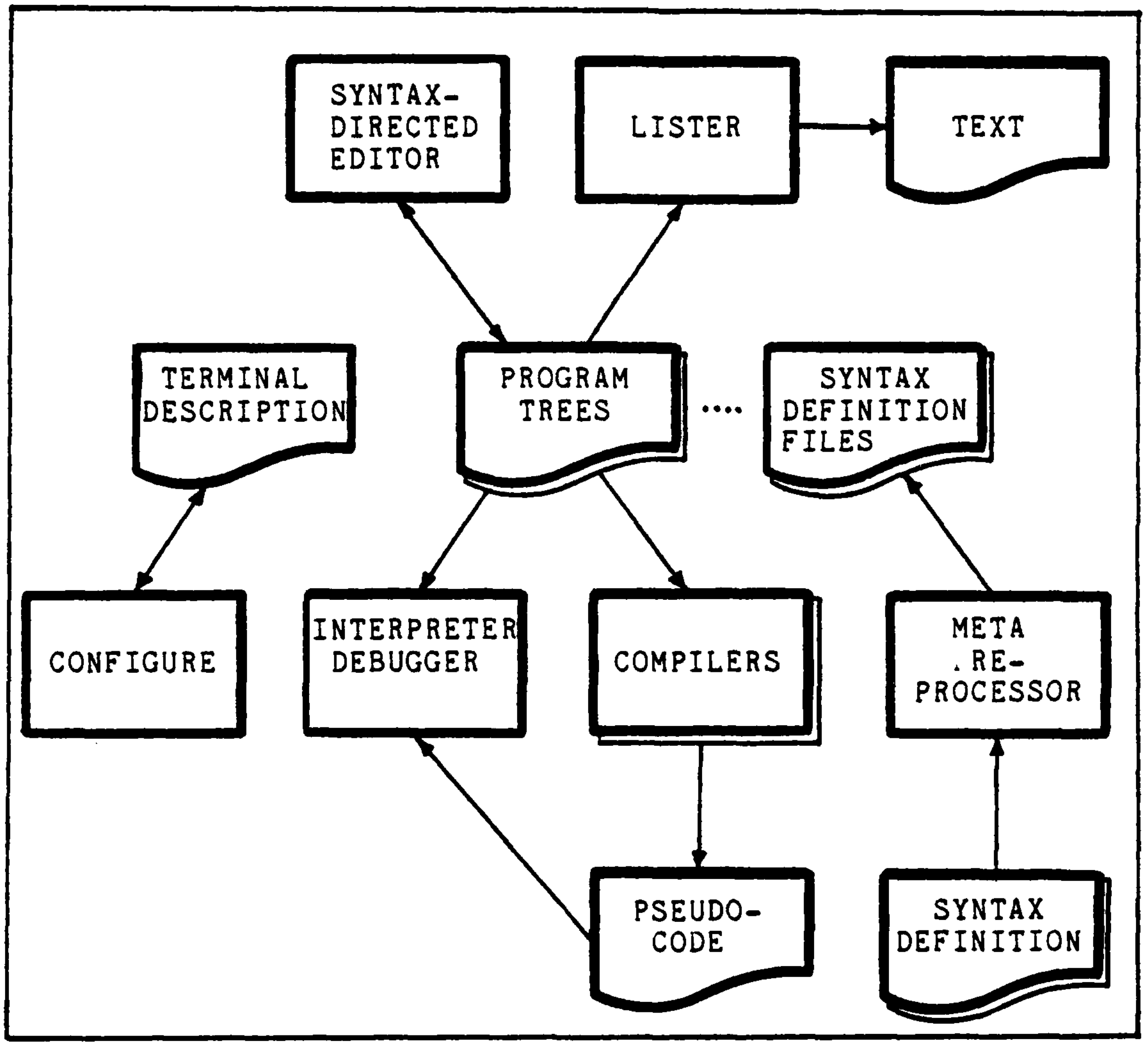


Figure 8-1. Prototype Environment Organization.

Four global software packages provide access to system data structures and display mechanisms for all the tools in the environment. The use of these packages greatly simplifies the creation of new tools within the environment.

8.2 Program Tree Access Package

The program tree access package provides any tool requiring access to program trees with primitives for most types of required tree operations including: attribute interrogation and modification, traversal, node creation and deletion, insertion and deletion, and subtree copying.

Significant problems are encountered on a small system when trying to manipulate a program tree too large to reside in available main memory. Virtual memory capabilities have been emulated to allow tree fragments to be loaded and unloaded from disc as required. Tree nodes are organized in a linear fashion in a file with pointers from node to node being relative from the start of the file. Relative pointers are used to ease the problems of moving absolute pointers around in memory. The program tree is divided into blocks to be swapped in and out on a least-recently-used basis. Machines providing virtual memory capabilities in hardware could significantly speed up tree access.

Each tree node occupies eight bytes. On the average, a program tree file will be up to eight times larger than the corresponding text-based program file. This program storage overhead is the tradeoff made to provide the capabilities of the syntax-directed environment. The decreasing price of secondary storage makes this overhead of less concern than in the past.

Accompanying each program tree a file information block which carries configuration management information along with areas for data exchange between environment tools and the name of the syntax description with which the program has been developed.

8.3 Syntax Description Access Package

A syntax description access package provides tools in the environment with primitives to access syntax descriptions for use in program display formatting and the analysis of syntactic content. The syntactic type of a program tree node and its relative position in a syntactic definition is represented by a pointer from the tree node into the syntax description.

The syntax description used by the environment is actually a condensed data structure produced from a META language definition by a preprocessor. The META preprocessor accepts a textual language definition, validates its format and contents, and creates a syntax description file for use in the environment.

8.4 Program Display Packages

The design strategy implemented in the prototype environment divides the processes of program image generation and display update into two distinct packages for use as general purpose environment utilities. The image generation package creates a desired display image of the program subtree in memory, completely independent of the editing functions and tree transformations taking place. This package also performs the automatic display justification and elision functions described in chapter 4, as well as determining the focus area to be highlighted on the display screen.

THE PROTOTYPE IMPLEMENTATION

The display update package performs screen functions to transform the current display image into the desired display image. The prototype system implements only a modest form of display update optimizations. More optimal display update algorithms, such as used in the CURSES package for UNIX (Ref 1), are readily available on other systems and should be easy to incorporate.

The display update process uses terminal-dependent codes for such functions as cursor positioning, erase to end of line and highlighting (reverse video). These codes are supplied to the environment in the form of a terminal description file created by a terminal configuration program in an interactive session with the user. The terminal description file also maintains screen height and width information and the terminal codes used to represent the various commands of the editor so that function keys or control keys available to the terminal may be used. The terminal configuration program allows the system to be used with most modern display terminals.

In order to remain an independent utility, the design of the program tree display package requires a regeneration of the entire display image after any change to the tree. This results in a gradual reduction in response time as the size of the display subtree increases. A more efficient means of display image generation might later improve system performance.

8.5 Environment Tool Interfaces

Use of the environment ordinarily begins with the editor. The editor when invoked is given the name of the program tree file to be edited. If this is a new file to be created, the name of the programming language must also be supplied. This directs the editor to the syntax description file for use in program tree synthesis.

At any time during the editing session the interpreter, compiler or lister may be invoked by command from the editor. These separate programs are loaded and given control after the program tree has been saved. The name of the language-dependent compiler to be loaded is automatically derived from the program tree's associated programming language name.

Errors discovered during processing by the compiler are marked for easy access by the editor. An array of tree node pointers is kept for this purpose in the program tree file's information block. The editing focus is automaticallv moved to the first error marker when control returns to the editor. Other error markers may also be examined with a simple editor command.

Nearly 7000 lines of C source code were generated during develɔpment of the syntax-directed environment. The syntax-directed editor program is approximately 24 kilobytes long; the compiler is nearly 28 kilobytes long.

9. THE ULTIMATE ENVIRONMENT

9.1 Incremental Compilation

The ultimate programming language environment should provide tools which will reduce the time spent in the development cycle between program modification and testing. The prototype syntax-directed environment attempts to do this by eliminating parsing analysis in the compiler and providing smooth interfaces between development tools.

The recompilation of program fragments which are unchanged from one iteration in the development cycle to the next wastes significant programmer time. Incremental compilation is one process by which such redundant compilations are avoided. A code data structure must be included which maps to the program tree so that the results of the compilation process may be retained. Changes or additions to program tree nodes are marked to trigger a later reexamination of their associated code fragments by the compiler. Some unchanged fragments within the scope of other changed fragments may also require reexamination. A type change in the declaration of a variable, for instance, might effect code generated elsewhere to access the variable. Since large portions of a program tree often remain unchanged, the time savings brought about by incremental compilation are anticipated to be significant. Such an incremental compilation environment has been successfully based on a syntax-directed editor system at Carnegie-Mellon University (Ref 5).

9.2 Multitasked Environment

The use of multitasking, which is supported by the ADA language, to support the programming environment is an attractive means of reducing unproductive time in the development cycle. The incremental compiler, as a separate but parallel task to the editor, could be triggered to recompile program fragments as they are changed. The interpreter/debugger might also be invoked to execute and debug code segments at any time during the editing process.

At this point all the tools in the environment actually merge into a single program development facility. The development cycle of program modification and testing collapses into a process where these tasks occur side by side. In an environment where these multiple tasks are handled by multiple processors, the user response time can be maintained at reasonable levels. As the capabilities of microprocessors increase and their prices decrease, such an environment becomes feasible and very attractive for individual programmer workstations and personal computers.

9.3 Semantic Specification

Means for specifying language semantics as an extension to a language syntax description have been developed elsewhere as a step toward the automatic generation of compilers from language descriptions (Ref 8). Such extensions might be applied to META grammar descriptions to provide the syntax-directed editor with enough information to disallow or flag semantic errors in the program tree. The specification of a language's code generation requirements should also be considered.

The analysis of semantic properties by the syntax-directed editor is by no means a trivial task. A single program change may affect the semantic validity of any number of other program fragments. Consider, for instance, the removal of a variable's declaration and the impact upon all its references. Such considerations forced this topic beyond the scope of this research.

10. CONCLUSIONS AND RECOMMENDATIONS

10.1 Conclusions

Several important advantages have been demonstrated by the prototype syntax-directed programming language environment. The syntactically correct program tree structure produced by the editor eliminates the need for a parser within the compiler. This reduces compilation time and makes compilers simpler and easier to build. Programmer time need no longer be wasted repairing syntax errors detected by the compiler. More effort can be devoted to improving program logic and design, with less effort spent struggling with language syntax.

The interpreter/debugger demonstrates how the program tree structure can be associated with other structures, such as code generated for the program, to provide powerful features, such as program display feedback to the user during execution. The development of a tool with these capabilities in a text-based environment is considerably more difficult.

Although the syntax-directed environment has been developed for the ADA language, any programming language may be incorporated. To process any additional language requires a compiler for the language and a META syntax description to define the language for the environment. All other tools are language independent, requiring no modification.

The entire prototype environment was implemented on a microcomputer system, indicating that complete syntax-directed program development facilities can be provided at relatively low cost. The introduction of large capacity mass storage and 16-bit microprocessor technology to microcomputers makes single-user programmer workstations quite attractive and affordable.

Syntax-directed editors should also have a significant impact upon education. Teachers of basic programming may place less emphasis on language syntax and more emphasis on sound programming techniques. The language subsetting feature of the environment may be used to introduce students to subsets of increasing size until the entire language has been presented.

10.2 Recommendations

Extensions to this research effort might proceed in many directions. The previous chapter has outlined several advanced concepts for major design extension of the environment. The introduction of incremental compilation is perhaps the best way of improving environment performance. Augmenting the META language with the ability to specify language semantics would allow for a much more useful editor and provide for automatic compiler generation facilities.

Moving the environment to a larger or more powerful machine might improve performance characteristics such as response time. The incorporation of multi-tasking to enhance the environment might best accomplished on the Intel iAPX-432 system or the VAX/780.

Another class of extensions to the prototype environment involve less drastic changes to the design of the existing implementation. The expansion of the prototype compiler toward a full ADA capability is of prime interest for use in teaching ADA and developing a complete ADA environment. Expansion of the environment debugging capabilities should accompany this effort to provide adequate support for more powerful language features.

As the editor is used, ways of improving the user interface will surely surface. New types of commands may be desirable and additional feedback in the form of help menus may prove useful. Modifications might also include device-independent support for advanced terminal capabilites such as the use of high-resolution graphics and color and input devices such as light pens or data tablets.

Support for additional languages, other than ADA, might also be of interest. Compiler development for such languages as PASCAL and C should be significantly simpler for the syntax-directed environment than for a text-based environment. Required META descriptions may be derived from other existing syntax definitions for these languages.

The requirements for ADA support environments specifies that tools be written in ADA where possible (Ref 2:15). The syntax-directed programming language environment should be translated into ADA as working ADA compilers become available. The software has been developed in modules intended to support the ADA packaging concept in anticipation of the need for such translation. Some attention should also be paid to the DOD requirements for configuration management and control for ADA support environments (Ref 2).

A standard intermediate representation form, called DIANA, has been developed for ADA programs (Ref 7). An additional tool could be built for the environment to produce DIANA notation almost directly from the program tree representation (and vice versa) for communication with other ADA environments.

Where the need arises to exchange programs with text-based environments, an additional tool will be required to produce program trees from a text image. Such a tool might be generated automatically from a syntax description in a manner similar to the language development aids YACC and LEX (Ref 9). The program lister of the syntax-directed environment may be used to perform the opposite task of generating text image files from a program tree.

Syntax-directed editors and environments surrounding them are finally being developed to offload mundane programmer tasks to the computer. It seems ironic that programmers, bound by inadequate and cumbersome development facilities, are among the last element of society to be liberated by computers.

BIBLIOGRAPHY

1. Arnold, Kenneth C. Screen Updating and Cursor Movement Optimization: A Library Package. Computer Science Division, University of California, Berkeley.

2. Department of Defense. Requirements for ADA Programming Support Environments. Washington, D.C., 1980.

3. Department of Defense. Reference Manual for the Ada Programming Language. Washington, D.C., 1980.

4. Dijkstra, Edsger W. "How Do We Tell Truths That Might Hurt?" ACM Sigplan Notices, 17 (5): 13-15, (May 1982).

5. Feiler, Peter H. and Raul Medina-Mora. An Incremental Programming Envirinment. Carnegie-Mellon Univ., Pittsburgh, PA. Dept. of Computer Science, April 1980.

6. Gaudino, Richard L. Analysis and Design of Interactive Debugging for the ADA Programming Support Environment. Masters Thesis, Air Force Institute of Technology, November 1981.

7. Goos, G. and Wm. A. Wulf. Diana Reference Manual. Institute Fuer Informatik II, March 1981.

8. Holt, R.C. "An Introduction to S/SL: Syntax/Semantic Language," ACM Transactions on Programming Languages and Systems, 4 (2) :149, (April 1982).

9. Johnson, S. C. and M. E. Lesk. "Language Development Tools," The Bell System Technical Journal, 57 (6) Part 2: 2155-2175 (July-August 1978).

10. MacLennan, Bruce J. The Automatic Generation of Syntax-Directed Editors. Naval Postgraduate School, Monterey, CA., 1981.

11. Shockley, William R. and Daniel P. Haddow. A Conceptual Framework for Grammar Driven Synthesis. Masters Thesis, Naval Postgraduate School, Monterey, CA., 1981.

12. Teitelbaum, Tim, et. al. "The Why and Wherefore of the Cornell Program Synthesizer," Proceedings of the ACM SIGPLAN SIGOA Symposium on Text Manipulation. (June 1981): 8-16.

13. Wirth, Nicklaus. Algorithms + Data Structures = Programs. Prentice-Hall, 1976.

14. Wirth, Nicklaus. "What Can We Do About the Unnecessary Diversity of Notation for Syntactic Definitions?" Communications of the ACM, 20 (11): 823 (November 1977).

APPENDIX I. META SYNTAX DESCRIPTION LANGUAGE

The following is a definition of the META syntax description language, given in META.

```
syntax =
        rule
        { @ rule } ;

rule =
        identifier ^ "="
                + definition ";" ;

identifier =
        'AZ|az' { '09|AZ|_|az' } ;

definition = <
        alternation
        concatenation >;

alternation =
        "<" ^ element { ^ element } ^ ">" ;

concatenation =
        term { ^ term } ;

element =
        primary [ ^ "!" ] [ ^ index ] ;

term = <
        option
        repeater
        primary >;

primary =
        [ '+|@|^' ^ ] factor [ ^ '^' ] ;

index =
        "$" { '07' } ;

option =
        "[" element "]" ;

repeater =
        "{" element "}" ;

factor = <
        identifier
        string
        set >;
```

```
string =
        """" { ' ~' } """" ;

set =
        "'" pair { pairs } "'" ;

pair =
        ' ~' [ ' ~' ] ;

pairs =
        "¦" pair ;
```

APPENDIX II. META DESCRIPTION FOR ADA

The following is a description of the ADA programming language, written in META, as adapted from the 1980 ADA Reference Manual.

```
compilation =
        compilation_unit
        { @ compilation_unit $0 } ;

compilation_unit = <
        proc_body
        func_body $0
        pack_body $0
        proc_decl $0
        func_decl $0
        pack_decl $0
        with_use_clause $0
        subunit $0
        pragma $0 >;

proc_body =
        proc_spec ^ "is"
                { + decl }
                { + rep_spec ! $0 }
                { + program_component }
        @ "begin"
                + seq_of_stmts
        [ @ exceptions ! $0 ]
        @ "end" [ ^ identifier ] ";" ;

func_body =
        func_spec ^ "is"
                { + decl }
                { + rep_spec ! }
                { + program_component }
        @ "begin"
                + seq_of_stmts
        [ @ exceptions ! ]
        @ "end" [ ^ designator ] ";" ;

pack_body =
        "package" ^ "body" ^ identifier ^ "is"
                { + decl }
                { + rep_spec ! }
                { + program_component }
        [ @ body_part ]
        @ "end" [ ^ identifier ] ";" ;
```

```
proc_decl = <
        proc_spec_semi
        generic_proc_decl
        generic_proc_instant >;

func_decl = <
        func_spec_semi
        generic_func_decl
        generic_func_instant >;

pack_decl = <
        pack_spec
        generic_pack_decl
        generic_pack_instant >;

with_use_clause =
        with_clause [ ^ use_clause ] ;

subunit =
        "separate" ^ "(" name ")" ^ subunit_body ;

pragma =
        "pragma" ^ identifier [ actual_param_part ] ";" ;

proc_spec =
        "procedure" ^ identifier [ formal_part $0 ] ;

decl = <
        object_decl
        type_decl $0
        subtype_decl $0
        number_decl $0
        func_decl $0
        proc_decl $0
        pack_decl $0
        task_decl $0
        exception_decl $0
        rename_object $0
        rename_exception $0
        rename_proc $0
        rename_func $0
        rename_pack $0
        rename_task $0
        use_clause $0
        pragma $0 >;

rep_spec = <
        length_spec
        enum_type_rep
        record_type_rep
        address_spec >;
```

```
program_component = <
        proc_body
        func_body $0
        pack_body $0
        task_body $0
        proc_stub $0
        func_stub $0
        pack_stub $0
        task_stub $0
        pack_decl $0
        task_decl $0 >;

seq_of_stmts =
        stmt
        { @ stmt } ;

exceptions =
        "exception"
                { + exception_handler } ;

identifier =
        'AZ|az' { '09|AZ|_|az' } ;

func_spec =
        "function" ^ designator [ ^ formal_part ]
                        ^ "return" ^ subtype_indication ;

designator = <
        identifier
        operator_symbol >;

body_part =
        "begin"
                + seq_of_stmts
        [ @ exceptions ! ] ;

proc_spec_semi =
        proc_spec ";" ;

generic_proc_decl =
        "generic"
                { + generic_formal_param }
        @ proc_spec ";" ;

generic_proc_instant =
        "procedure" ^ identifier ^ "is"
                             ^ generic_instant ";" ;

func_spec_semi =
        func_spec ";" ;
```

```
generic_func_decl =
        "generic"
                { + generic_formal_param }
        @ func_spec ";" ;

generic_func_instant =
        "function" ^ designator ^ "is"
                              ^ generic_instant ";" ;

pack_spec =
        "package" ^ identifier ^ "is"
                { + decl }
        [ @ private_part ]
        @ "end" [ ^ identifier ] ";" ;

generic_pack_decl =
        "generic"
                { + generic_formal_param }
        @ pack_spec ;

generic_pack_instant =
        "package" ^ identifier ^ "is" ^ generic_instant ";";

with_clause =
        "with" ^ name { names } ";" ;

use_clause =
        "use" ^ name { names } ";" ;

name = <
        identifier
        indexed_component $0
        selected_component $0
        slice $0
        attribute $0
        func_call $0
        operator_symbol $0 >;

subunit_body = <
        proc_body
        func_body
        pack_body
        task_body >;

actual_param_part =
        "(" param_assoc { param_assocs } ")" ;

formal_part =
        "(" param_decl { param_decls } ")" ;

object_decl =
        id_list ":" [ ^ "constant" ] ^ object_type
                                 [ ^ initial ] ";" ;
```

```
type_decl =
        "type" ^ identifier [ ^ discrim_part ! ]
                                  [ ^ type_body ] ";" ;

subtype_decl =
        "subtype" ^ identifier ^ "is" ^
                                  subtype_indication ";" ;

number_decl =
        id_list ":" ^ "constant" ^ initial ";" ;

task_decl =
        "task" [ ^ "type" ] ^ identifier [ ^ task_def ] ";";

exception_decl =
        id_list ":" ^ "exception" ";" ;

rename_object =
        identifier ":" ^ name ^ "renames" ^ name ";" ;

rename_exception =
        identifier ":" ^ "exception" ^ "renames" ^ name ";";

rename_proc =
        proc_spec ^ "renames" ^ name ";" ;

rename_func =
        func_spec ^ "renames" ^ name ";" ;

rename_pack =
        "package" ^ identifier ^ "renames" ^ name ";" ;

rename_task =
        "task" ^ identifier ^ "renames" ^ name ";" ;

length_spec =
        "for" ^ attribute ^ "use" ^ expression ";" ;

enum_type_rep =
        "for" ^ name ^ "use" ^ aggregate ";" ;

record_type_rep =
        "for" ^ name ^ "use"
                + record_rep ;

address_spec =
        "for" ^ name ^ "use" ^ "at" ^ simple_exp ";" ;
```

```
task_body =
        "task" ^ "body" ^ identifier ^ "is"
                { + decl }
                { + rep_spec ! }
                { + program_component }
        @ "begin"
                + seq_of_stmts
        [ @ exceptions ! ]
        @ "end" [ ^ identifier ] ";" ;

proc_stub =
        proc_spec ^ "is" ^ "separate" ";" ;

func_stub =
        func_spec ^ "is" ^ "separate" ";" ;

pack_stub =
        "package" ^ "body" ^ identifier ^ "is" ^
                                "separate" ";" ;

task_stub =
        "task" ^ "body" ^ identifier ^ "is" ^
                                "separate" ";" ;

stmt =
        { label ^ ! $0 } simple_stmt ;

exception_handler =
        "when" ^ exception_choice { exception_choices } ^
                                "=>"
                + seq_of_stmts ;

subtype_indication =
        name [ ^ constraint $0 ] ;

operator_symbol =
        char_string ;

generic_formal_param = <
        param_decl_semi
        generic_proc
        generic_func
        generic_type >;

generic_instant =
        "new" ^ name [ generic_assocs ] ;

private_part =
        "private"
                { + decl }
                { + rep_spec ! } ;

names =
        "," ^ name ;
```

```
indexed_component =
        name "(" expression { expressions } ")" ;

selected_component =
        name "." component ;

slice =
        name "(" discrete_range ")" ;

attribute =
        name "'" identifier ;

func_call =
        name [ actual_param_part ] ;

param_assoc =
        [ param_link ^ ] expression ;


param_assocs =
        "," ^ param_assoc ;

param_decl =
        id_list ":" [ ^ "in" ] [ ^ "out" ]
                    ^ subtype_indication [ ^ initial ] ;

param_decls =
        ";" ^ param_decl ;

id_list =
        identifier { identifiers } ;

object_type = <
        subtype_indication
        array_type_def $0 >;

initial =
        ":=" ^ expression ;

discrim_part =
        "(" discrim_decl { discrim_decls } ")" ;

type_body =
        "is" ^ type_def ;

task_def =
        "is"
                { + entry_decl }
                { + rep_spec ! }
        @ "end" [ ^ identifier ] ;
```

```
expression = <
        relation
        and_comp
        or_comp
        and_then_comp $0
        or_else_comp $0
        xor_comp $0 >;

aggregate =
        "(" component_assoc { component_assocs } ")" ;

record_rep =
        "record" [ ^ align_clause ]
                { + name_location }
        @ "end" ^ "record" ";" ;

simple_exp =
        [ unary_operator ! ] term { terms } ;

label =
        "<<" identifier ">>" ;

simple_stmt = <
        assignment_stmt
        if_stmt
        loop_stmt
        proc_call
        case_stmt $0
        block $0
        exit_stmt $0
        return_stmt $0
        goto_stmt $0
        entry_call $0
        delay_stmt $0
        abort_stmt $0
        raise_stmt $0
        code_stmt $0
        accept_stmt $0
        selective_wait $0
        cond_entry_call $0
        timed_entry_call $0
        null_stmt $0 >;

exception_choice = <
        name
        "others" >;

exception_choices =
        "|" ^ exception_choice ;
```

```
constraint = <
        range_constraint
        float_pt_constraint
        fixed_pt_constraint
        index_constraint
        discrim_constraint >;

char_string =
        """" { ' ~' } """" ;

param_decl_semi =
        param_decl ";" ;

generic_proc =
        "with" ^ proc_spec [ ^ generic_is ] ";" ;

generic_func =
        "with" ^ func_spec [ ^ generic_is ] ";" ;

generic_type =
        "type" ^ identifier [ ^ discrim_part ] ^ "is"
                                   ^ generic_type_def ";" ;

generic_assocs =
        "(" generic_assoc { generic_assoc } ")" ;

expressions =
        "," ^ expression ;

component = <
        identifier
        "all"
        operator_symbol >;

discrete_range = <
        type_range
        range >;

param_link =
        identifier ^ "=>" ;

identifiers =
        "," ^ identifier ;

array_type_def = <
        constrained_array
        unconstrained_array >;

discrim_decl =
        id_list ":" ^ subtype_indication [ ^ initial ] ;

discrim_decls =
        ";" ^ discrim_decl ;
```

```
type_def = <
        range_constraint
        float_pt_constraint
        fixed_pt_constraint
        array_type_def
        record_type_def
        enum_type_def
        access_type_def
        derived_type_def
        private_type_def >;

entry_decl =
        "entry" ^ identifier [ ^ entry_dimension ]
                                  [ ^ formal_part ] ";" ;

relation =
        simple_exp [ ^ relation_part ! ] ;

and_comp =
        relation ^ { and_relation } ;

or_comp =
        relation ^ { or_relation } ;

and_then_comp =
        relation ^ { and_then_relation } ;

or_else_comp =
        relation ^ { or_else_relation } ;

xor_comp =
        relation ^ { xor_relation } ;

component_assoc =
        [ choice_link ^ ] expression ;

component_assocs =
        "," ^ component_assoc ;

align_clause =
        "at" ^ "mod" ^ simple_exp ";" ;

name_location =
        name ^ "at" ^ simple_exp ^ "range" ^ range ";" ;

unary_operator = <
        "+"
        "-"
        "not" >;

term =
        factor { factors } ;
```

```
terms =
        add_op term ;

assignment_stmt =
        name ^ ":=" ^ expression ";" ;

if_stmt =
        "if" ^ expression ^ "then"
                + seq_of_stmts
        { @ elsif_part }
        [ @ else_part ]
        @ "end" ^ "if" ";" ;

loop_stmt =
        [ tag ^ ! $0 ] [ iteration_clause ^ ] "loop"
                + seq_of_stmts
        @ "end" ^ "loop" [ ^ identifier ! $0 ] ";" ;

proc_call =
        name [ actual_param_part $0 ] ";" ;

case_stmt =
        "case" ^ expression ^ "is"
                { + cases }
        @ "end" ^ "case" ";" ;

block =
        [ tag ^ ! ] [ declare ]
        @ "begin"
                + seq_of_stmts
        [ @ exceptions ! ]
        @ "end" [ ^ identifier ! ] ";" ;

exit_stmt =
        "exit" [ ^ name ] [ ^ when_clause ] ";" ;

return_stmt =
        "return" [ ^ expression ] ";" ;

goto_stmt =
        "goto" ^ name ";" ;

entry_call =
        name [ actual_param_part ] ";" ;

delay_stmt =
        "delay" ^ simple_exp ";" ;

abort_stmt =
        "abort" ^ name { names } ";" ;

raise_stmt =
        "raise" [ ^ name ] ";" ;
```

```
code_stmt =
        qualified_exp ";" ;

accept_stmt =
        "accept" ^ name [ formal_part ]
                              [ ^ accept_action ] ";" ;

selective_wait =
        "select" [ condition_link ]
                + select_alternative
        { @ or_clause }
        [ @ else_part ]
        @ "end" ^ "select" ";" ;

cond_entry_call =
        "select"
                + entry_call
                [ + seq_of_stmts ]
        @ "else"
                + seq_of_stmts
        @ "end" ^ "select" ";" ;

timed_entry_call =
        "select"
                + entry_call
                [ + seq_of_stmts ]
        @ "or"
                + delay_alternative
        @ "end" ^ "select" ";" ;

null_stmt =
        "null" ";" ;

range_constraint =
        "range" ^ range ;

float_pt_constraint =
        "digits" ^ simple_exp [ ^ range_constraint ] ;

fixed_pt_constraint =
        "delta" ^ simple_exp [ ^ range_constraint ] ;

index_constraint =
        "(" discrete_range { discrete_ranges } ")" ;

discrim_constraint =
        "(" discrim_spec { discrim_specs } ")" ;

generic_is =
        "is" ^ generic_name ;
```

```
generic_type_def = <
        generic_discrete
        generic_integer
        generic_float
        generic_fixed
        array_type_def
        access_type_def
        private_type_def >;

generic_assoc =
        [ param_link ^ ] generic_actual_param ;

type_range =
        name [ ^ range_constraint ] ;

range =
        simple_exp ".." simple_exp ;

constrained_array =
        "array" ^ index_constraint ^ "of"
                                   ^ subtype_indication ;

unconstrained_array =
        "array" ^ "(" index { indices } ")" ^ "of"
                                   ^ subtype_indication ;

record_type_def =
        "record"
                + component_list
        @ "end" ^ "record" ;

enum_type_def =
        "(" enum_lit { enum_lits } ")" ;

access_type_def =
        "access" ^ subtype_indication ;

derived_type_def =
        "new" ^ subtype_indication ;

private_type_def =
        [ "limited" ^ ] "private" ;

entry_dimension =
        "(" discrete_range ")" ;

relation_part = <
        relational
        in_range $0 >;

and_relation =
        "and" ^ relation ;
```

```
or_relation =
        "or" ^ relation ;

and_then_relation =
        "and" ^ "then" ^ relation ;

or_else_relation =
        "or" ^ "else" ^ relation ;

xor_relation =
        "xor" ^ relation ;

choice_link =
        choice { choices } "=>" ;

factor =
        primary [ power ! $0 ] ;

factors =
        mul_op factor ;

add_op = <
        "+"
        "-"
        "&" $0 >;

elsif_part =
        "elsif" ^ expression ^ "then"
                + seq_of_stmts ;

else_part =
        "else"
                + seq_of_stmts ;

tag =
        identifier ":" ;

iteration_clause = <
        while_clause
        for_clause $0 >;

cases =
        "when" choice { choices } "=>"
                + seq_of_stmts ;

declare =
        "declare"
                { + decl }
                { + rep_spec ! }
                { + program_component } ;

when_clause =
        "when" ^ expression ;
```

```
qualified_exp =
        name "'" agg_or_exp ;

accept_action =
        "do"
                + seq_of_stmts
        @ "end" [ ^ identifier ] ;

condition_link =
        "when" ^ expression ^ "=>" ;

select_alternative = <
        accept_alternative
        delay_alternative
        terminate >;

or_clause =
        "or" [ ^ condition_link ]
                + select_alternative ;

delay_alternative =
        delay_stmt
        [ @ seq_of_stmts ] ;

discrete_ranges =
        "," ^ discrete_range ;

discrim_spec =
        [ discrim_link ^ ] expression ;

discrim_specs =
        "," ^ discrim_spec ;

generic_name = <
        name
        "<>" >;

generic_discrete =
        "(" "<>" ")" ;

generic_integer =
        "range" ^ "<>" ;

generic_float =
        "delta" ^ "<>" ;

generic_fixed =
        "digits" ^ "<>" ;

generic_actual_param = <
        expression
        name
        subtype_indication >;
```

META DESCRIPTION FOR ADA

```
index =
        name ^ "range" ^ "<>" ;

indices =
        "," ^ index ;

component_list = <
        components
        null_comp >;

enum_lit = <
        identifier
        char_lit >;

enum_lits =
        "," ^ enum_lit ;

relational =
        rel_op ^ simple_exp ;

in_range =
        simple_exp [ ^ "not" ] ^ "in" ^ range_or_subtype ;

choice = <
        simple_exp
        discrete_range
        "others" >;

choices =
        "|" ^ choice ;

primary = <
        decimal_number
        name
        nested_exp
        based_number $0
        enum_lit $0
        char_string $0
        func_call $0
        "null" $0
        aggregate $0
        allocator $0
        type_conversion $0
        qualified_exp $0 >;

power =
        "**" primary ;

mul_op = <
        "*"
        "/"
        "mod" $0
        "rem" $0 >;
```

```
while_clause =
        "while" ^ expression ;

for_clause =
        "for" ^ identifier ^ "in" [ ^ "reverse" ! ]
                                    ^ discrete_range ;

agg_or_exp = <
        aggregate
        nested_exp >;

accept_alternative =
        accept_stmt
        [ @ seq_of_stmts ] ;

terminate =
        "terminate" ";" ;

discrim_link =
        name { names } ^ "=>" ;

components =
        { @ component_decl }
        [ @ variant_part ] "" ;

null_comp =
        "null" ";" ;

char_lit =
        "'" ' ~' "'" ;

rel_op = <
        "="
        "/="
        "< "
        "<="
        "> "
        ">=" >;

range_or_subtype = <
        range
        subtype_indication >;

decimal_number =
        integer [ decimal_part ! $0 ] [ exponent ! $0 ] ;

nested_exp =
        "(" expression ")" ;

based_number =
        integer "#" based_integer [ based_decimal ! ]
                           "#" [ exponent ! ] ;
```

META DESCRIPTION FOR ADA

```
allocator =
        "new" ^ name [ ^ allocation ] ;

type_conversion =
        name "(" expression ")" ;

component_decl =
        id_list ":" ^ object_type [ ^ initial ] ";" ;

variant_part =
        "case" ^ name ^ "is"
                { + variant_case }
        @ "end" ^ "case" ";" ;

integer =
        '09' { '09|_' } ;

decimal_part =
        "." integer ;

exponent =
        "E" [ sign ] integer ;

based_integer =
        '09|AZ|az' { '09|AZ|_|az' } ;

based_decimal =
        "." based_integer ;

allocation = <
        nested_exp
        aggregate
        discrim_constraint
        index_constraint >;

variant_case =
        "when" ^ choice { choices } ^ "=>"
                + component_list ;

sign = <
        "+"
        "-" >;
```

III. META DESCRIPTION FOR ADAO

The following is a META description for the ADAO subset implemented in the full prototype syntax-directed programming language environment. This subset is the $0 subset extracted from the previous ADA description.

```
compilation =
        compilation_unit ;

compilation_unit = <
        proc_body >;

proc_body =
        proc_spec ^ "is"
                { + decl }
                { + program_component }
        @ "begin"
                + seq_of_stmts
        @ "end" [ ^ identifier ] ";" ;

proc_spec =
        "procedure" ^ identifier ;

decl = <
        object_decl >;

program_component = <
        proc_body >;

seq_of_stmts =
        stmt
        { @ stmt } ;

identifier =
        'AZ¦az' { '09¦AZ¦_¦az' } ;

object_decl =
        id_list ":" [ ^ "constant" ] ^ object_type
                                           [ ^ initial ] ";" ;

stmt =
        simple_stmt ;

id_list =
        identifier { identifiers } ;

object_type = <
        subtype_indication >;
```

META DESCRIPTION FOR ADAO

```
initial =
        ":=" ^ expression ;

simple_stmt = <
        assignment_stmt
        if_stmt
        loop_stmt
        proc_call >;

identifiers =
        "," ^ identifier ;

subtype_indication =
        name ;

expression = <
        relation
        and_comp
        or_comp >;

assignment_stmt =
        name ^ ":=" ^ expression ";" ;

if_stmt =
        "if" ^ expression ^ "then"
                + seq_of_stmts
        { @ elsif_part }
        [ @ else_part ]
        @ "end" ^ "if" ";" ;

loop_stmt =
        [ iteration_clause ^ ] "loop"
                + seq_of_stmts
        @ "end" ^ "loop" ";" ;

proc_call =
        name ";" ;

name = <
        identifier >;

relation =
        simple_exp [ ^ relation_part ! ] ;

and_comp =
        relation ^ { and_relation } ;

or_comp =
        relation ^ { or_relation } ;

elsif_part =
        "elsif" ^ expression ^ "then"
                + seq_of_stmts ;
```

```
else_part =
        "else"
                + seq_of_stmts ;

iteration_clause = <
        while_clause >;

simple_exp =
        [ unary_operator ! ] term { terms } ;

relation_part = <
        relational >;

and_relation =
        "and" ^ relation ;

or_relation =
        "or" ^ relation ;

while_clause =
        "while" ^ expression ;

unary_operator = <
        "+"
        "-"
        "not" >;

term =
        factor { factors } ;

terms =
        add_op term ;

relational =
        rel_op ^ simple_exp ;

factor =
        primary ;

factors =
        mul_op factor ;

add_op = <
        "+"
        "-" >;

rel_op = <
        "="
        "/="
        "< "
        "<="
        "> "
        ">=" >;
```

META DESCRIPTION FOR ADAO

```
primary = <
        decimal_number
        name
        nested_exp >;

mul_op = <
        "*"
        "/" >;

decimal_number =
        integer ;

nested_exp =
        "(" expression ")" ;

integer =
        '09' { '09|_' } ;
```

APPENDIX IV. SYSTEM USER'S MANUAL

SYSTEM USER'S MANUAL

FOR THE SYNTAX-DIRECTED PROGRAMMING LANGUAGE ENVIRONMENT

User's Manual Contents

1. SYNTAX-DIRECTED EDITOR

The syntax-directed editor provides the capability to create and modify programs under programmer direction. The editor is responsible for maintaining the syntactic validity of the program tree at all times. The editor also coordinates the use of the other support tools in the environment, such as the compiler and interpreter. For program development, therefore, the user need only be concerned with the use of the editor.

1.1 Entering the Editor

The editor is invoked by its name, SYNDE, followed by the filename of the program to be edited. If the specified file does not exist (it is a new file to be created by the editor), the filename must be followed by a programming language name. This name directs the editor to a syntax description file (which must be present on disc) to guide program synthesis. As an example:

```
SYNDE TEST ADAO
```

would be used to create a new program, called TEST, in the ADAO language subset using the syntax description file named ADAO.SDF.

```
SYNDE TEST
```

would be used to edit the previously created TEST program.

1.2 Editor Commands

The keystroke sequences used to invoke particular editor commands are specified by the user during terminal configuration. Terminal configuration is described in chapter 2 of this appendix.

The entire program subtree rooted at the focus designates the entity being considered by the editor for manipulation by the next command. The focus image is highlighted on the display screen, typically by reverse video or higher character intensity. The means of highlighting is determined during terminal configuration.

The editing commands are grouped into three categories: focus movement, edit commands and control commands. Each command is responded to either by its effect on the display of the program tree or with an appropriate display message.

1.2.1 Focus Movement

Focus movement to adjacent program tree nodes such as parent, son or sibling form the primary type of movement within the program tree.

1.2.1.1 Move Right

The MOVE RIGHT command moves the focus to the right sibling of the focus. If no right sibling exists, the tree is ascended until some ancestor is found with a right sibling which becomes the focus. This process attempts to maintain movement in the direction of an inorder traversal of the tree.

1.2.1.2 Move Left

The MOVE LEFT command moves the focus to the left sibling of the focus. If no left sibling exists, the tree is ascended until some ancestor is found with a left sibling which becomes the focus. This process attempts to maintain movement in the reverse direction of an inorder traversal of the tree.

1.2.1.3 Move Long Right

The MOVE LONG RIGHT command moves the focus to the right and past any siblings which are generated from the same repetition element in a production.

1.2.1.4 Move Long Left

The MOVE LONG LEFT command moves the focus to the left and past any siblings which are generated from the same repetition element in a production.

1.2.1.5 Move Up

The MOVE UP command moves the focus to the parent of the focus, if one exists. This raises the focus to a higher syntactic level. If the new focus has children other than the previous focus it will also have a larger frontier. In this case, the extended cursor designating the range of the focus subtree will enlarge to encompass a greater amount of the program. This corresponds to a "zoom-out" effect.

1.2.1.6 Move Down

The MOVE DOWN command moves the focus to the son of the focus, if one exists. This drops the focus to a lower syntactic level. If the previous focus has children other than the new focus, the extended cursor will shrink to encompass a smaller amount of the program. This corresponds to a "zoom-in" effect.

1.2.1.7 Move Long Up

The MOVE LONG UP command is equivalent to a series of move up commands. The focus is moved up the tree until its frontier becomes larger. The purpose is to force the type of "zoom-out" effect described above and to increase the size of the program fragment designated by the focus.

1.2.1.8 Move Long Down

The MOVE LONG DOWN command is equivalent to a series of move down commands. It serves to force the "zoom-in" effect in contrast to the MOVE LONG UP command.

1.2.1.9 Move To Leaf

The MOVE TO LEAF command descends the tree from the current focus, following an inorder traversal to the first leaf node. This is the quickest way to get to the bottom of the tree where most of the initial program entry occurs.

1.2.1.10 Move To Last Focus

The previous focus location is saved by editor commands which change the focus. A return to the previous focus is then accomplished by the MOVE TO LAST FOCUS command.

1.2.1.11 Mark and Go

Ten markers (pointers into the tree) are kept in the file information block for a program tree. The MARK command may be used to clear an existing marker at the focus or to set any marker zero through four to point to the focus. Markers five through nine are reserved by the system to mark errors detected by other tools in the environment, such as the compiler. The GO command may be used to move the focus to any requested marker that is set. The GO command may also be used to move the focus to the root of the program tree or to the root of a clipping tree which is used for cut and paste operations described below. Markers are preserved in the tree from one editing session to another.

1.2.2 Edit Commands

1.2.2.1 Insert Right

The INSERT RIGHT command may be used to insert a conditional element as the right sibling of the focus. The type of the element to be synthesized into the tree is determined from the production definition of the parent of the focus. A valid conditional element must be capable of being inserted at that point. An optional element must not have already been synthesized into the list since only one is allowed. Inserting a second repeater immediately behind an identical, unestablished repeater is disallowed as a useless, although syntactically valid, operation.

If two or more successive conditional elements exist in the production, the first may need to be inserted before the second may be inserted. This results from the need to place the focus on the node representing the element preceeding the one which is to be inserted. If this node does not exist, it must be inserted, but may be deleted after its use.

1.2.2.2 Insert Left

The INSERT LEFT command is the equivalent of the INSERT RIGHT command for inserting a conditional element as the left sibling of the focus. This command is required to insert a conditional to the left of the leftmost sibling.

1.2.2.3 Clip

The CLIP command copies the subtree designated by the focus to a "clipping" tree. Any previous clipping is discarded and the program tree remains unchanged. This represents the "cut" portion of a "cut and paste" operation. A pointer to the clipping tree is maintained in the file information block which accompanies the program tree. The clipping tree remains with the program tree during its lifetime until replaced by another clip type operation. The clipping tree may itself be edited by using the GO command to move the focus into the clipping tree.

As a safety precaution, to preserve the clipping tree from inadvertant loss, the clip operation is not allowed and the clipping tree remains unchanged if the focus is a leaf node. This would not be a significant operation since leaf nodes are degenerate subtrees and are easily regenerated.

1.2.2.4 Copy

The COPY command may be used to attach a copy of the clipping tree to the program tree at the focus node. The previous focus subtree is discared and is lost, without recovery. The clipping tree remains unchanged. This represents the "paste" portion of a "cut and paste" operation. The copy operation is allowed only if the syntactic type of the root of the clipping is the same as that of the focus, or if the focus represents a non-terminal with an alternation definition and the root of the clipping is one of its alternatives. Here the clipping is attached below the focus rather than at the focus. The syntactic type of the focus is displayed in the header line of the main window. The syntactic type of the clipping may be observed by requesting display of the clipping in the second window with the window command.

1.2.2.5 Kill

The KILL command may be used to delete the focus from the program. If the focus represents a conditional element, it is simply deleted. Otherwise the node is required for syntactic validity. In this case, its sons and their subtrees are deleted. If the remaining leaf node represents a non-terminal with a concatenation definition, the node is re-expanded producing a new template. No recovery is possible from the KILL command. The DELETE command should be used where recovery is desired.

1.2.2.6 Delete

The DELETE command is equivalent to a CLIP command followed by a KILL command. The focus is copied to the clipping tree and is then deleted from the main program tree. Material lost from accidental deletes may be recovered with the copy command. As in the clip command, the focus is not copied if it is only a leaf node. This command is disallowed when the focus is in the clipping subtree itself.

1.2.3 Control Commands

The control commands provide the user with control over program display and interface to other environment programming tools.

1.2.3.1 Help

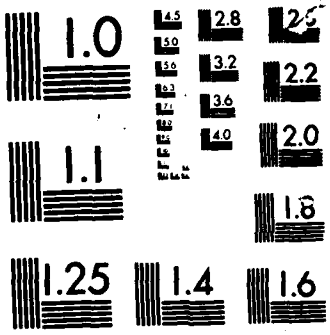
The HELP command toggles the users request for help menus. Help defaults to on at the start of the editing session. The only help menu currently implemented is the list of alternatives available for selection when the focus is at a leaf which is an alternation node.

1.2.3.2 Elide

The ELIDE command toggles the elide flag for the focus node. If the node was not previously elided, the portion of the program tree contained in the focus becomes the extent

AD-A124 843 A SYNTAX-DIRECTED PROGRAMMING ENVIRONMENT FOR THE ADA PROGRAMMING LANGUAGE(U) AIR FORCE INST OF TECH WRIGHT-PATTERSON AFB OH SCHOOL OF ENGINEERING 2/2

UNCLASSIFIED S E FERGUSON DEC 82 AFIT/GCS/MA/82D-1 F/G 9/2 NL


1.0
2.8
3.2
2.2
3.6
4.0
2.0
1.1
1.8
1.25
1.4
1.6

of the program image displayed on the screen. Upon moving up the tree from the elided node, the display of its subtree is suppressed and replaced with a string supplied by the user during terminal configuration.

1.2.3.3 Window

The WINDOW command is used to control the presence of a second window. The window may be requested to display the clipping tree or any of the subtrees pointed to by markers zero through nine or to be closed entirely. The focus may not be moved to the second window, it is intended for display only, not editing. The window will, however, reflect any changes made to its displayed subtree that are accomplished in the main window. The window is automatically closed if the subtree being displayed is deleted by some edit command in the main window. The syntactic type of the root of the displayed subtree is given in the header line of the window.

1.2.3.4 Invoke Compiler

The language specific compiler for the program tree being edited is loaded and executed. The compiler flags any errors using program tree markers five through nine and will halt when all markers have been used or when compilation is complete. After compilation, execution returns to the editor and the focus is set to the node addressed by error marker 5, if set, or to the root of the program tree.

1.2.3.5 Invoke Interpreter

The interpreter is loaded and executed to process the program tree being edited. The interpreter compiles the program to create pseudo-code for interpretation. If errors are encountered, return to the editor follows the same process as for invocation of the compiler.

Before each instruction is executed, the topmost elements of the runtime stack are displayed along with the next instruction to be executed. The interpreter commands available to the user are explained below.

1.2.3.5.1 Single Step

The SINGLE STEP command, invoked by 'S' or SPACE causes execution of the displayed instruction. If this instruction is the last in the program, the interpreter is reset to resume execution at the start of the program and the stack is returned to the initial state. The focus of the display is moved to the tree node associated with the instruction.

1.2.3.5.2 Continue Execution

The CONTINUE EXECUTION command is invoked with 'C'. Execution of the program is allowed to continue until interrupted by another command. Each instruction execution is accompanied by update of the program tree display and dump of the stack and next instruction. Execution recycles to the start of the program after reaching the end.

1.2.3.5.3 Restart

The RESTART command, invoked by 'R', resets the interpreter to resume execution at the start of the program. The stack is returned to the initial state.

1.2.3.5.4 Exit Interpreter

The EXIT INTERPRETER command, invoked by 'E', returns control to the editor to resume editing of the program tree.

1.2.3.6 Invoke Lister

The language independent lister is loaded and executed to create a text image of the program tree being edited. The program lister produces a text format source file from the program tree for use in generating hardcopy listings or for transfer to a text-based environment. The text image is generated with unestablished conditional nodes omitted, since these are ignored by the compiler. Elided subtrees are not suppressed from the image. Control returns to the editor.

1.2.3.7 Exit Editor

The program tree being edited is saved and the editing session is terminated.

1.3 Program Leaf Mutations

The leaves of the tree are points at which user decisions are made to extend the program. Alternatives are selected, conditional nodes are established into the tree and character values for sets are supplied.

1.3.1 Alternative Selection

A leaf node which has an alternation definition requires the selection of one of its alternatives. If help is toggled on, the list of alternatives is presented at the bottom of the screen. The user merely types the name of the desired alternative. Command-completion by the editor speeds the selection and reduces the number of required keystrokes by determining the portion of the name common to all remaining alternatives. The alternatives for the ADA subunit, for example, are proc_body, func_body, pack_body and task_body. After typing 'p', the list is reduced to proc_body and pack_body. An 'r', then, reduces the list to the single alternative of proc_body.

1.3.2 Set selection

A leaf node which represents a set may accept a character value according to the range of characters given by its name. That name also represents the node's syntactic type which is displayed in the header line of the display window, providing the user with a range of valid characters. The value of any valid character typed will then be saved in the node and represent it in the program display image.

1.3.3 Conditional Node Establishment

Conditional leaf nodes require user interaction to establish them into the tree. If the node represents an alternation or a set, this interaction is implied by selecting an alternative or typing a character, respectively, as described above. If the node represents a concatenation non-terminal or a string, typing the first character of its displayed name will establish the node.

When established, the node's surrounding brackets or braces are no longer displayed. If the conditional being established is a repeater, a new, unestablished repeater of the same type is inserted after the focus in anticipation of the user's desire to later establish another such node in the tree.

2. TERMINAL CONFIGURATION

A configuration program, CONFIG, is used during installation to interactively prompt the user for information which is supplied to the system in a terminal description file. This file includes the screen dimensions and terminal strings to accomplish a variety of terminal functions. The terminal description file also contains the list of strings to be interpreted as specific commands by the editor so that they may be tailored to the control key or function key capabilities of a given terminal.

CONFIG may be invoked with:

```
CONFIG *
```

where the "*" is optional and specifies that the existing terminal description file (TERMINAL.TDF) is to be cleared before use. The user is given the opportunity to modify the recorded terminal characteristics and input command sequences.

CONFIG will inquire at times if the user wants to modify or reexamine portions of the terminal description. Respond to questions with a "Y" or "y" for yes or any other character for no.

CONFIG will first request information on the terminal's lines per screen, characters per line and number of lines to be used in the second window. CONFIG will prompt the user for numeric information by supplying the current value and an inclusive range of valid values. Enter a new value or type RETURN to retain the previous value. Invalid values are rejected.

Next, CONFIG will request input sequences to represent input commands. CONFIG presents the user with the current sequence (if one is present) and the option to change it or proceed to the next item. Control keys in a sequence are displayed as ^x where x is the control character plus a bias of 64 to produce a printable character. For example, a line-feed character (ASCII value 10) will display as ^J. CONFIG provides several opportunities during terminal configuration to reexamine character sequences and correct mistakes. When entering character sequences for input commands, merely strike the keys desired to invoke the command function. Input commands must begin with a non-printable control character. Each input command sequence must, of course, be unique.

CONFIG will then request output sequences to perform varies display funtions. These are entered in the same manner as input sequences. These sequences are:

- intialize terminal: Sent to the terminal at the start of the editing session to allow for special setup.

- display tab: Used for program indentation. This should contain only printable characters, usually a number of spaces.
- mark elided material: Used to represent an elided program subtree on the display.
- divide windows: A single character used to fill the header line of each window.
- clear screen: Clears the terminal screen.
- position cursor: Prefix of display command to position the cursor on the screen. This sequence will be followed by the display line (first is 0) plus 32 and the display column plus 32. Other forms of display cursor addressing are not yet supported by terminal configuration.
- erase to end of line: Clears the display from the cursor location to the end of the line.
- enter reverse video mode: Sets the terminal into a highlight mode, such as reverse video, to distinguish the focus.
- exit reverse video mode: Exit the display mode set by the "enter reverse video mode" sequence.
- terminate terminal: Sent to the terminal at the end of the editing session to clean up the terminal status.

3. LANGUAGE SYNTAX DESCRIPTION

3.1 The META Preprocessor

The syntax description actually used by the environment is a condensed representation of a textual META syntax description. The META preprocessor is invoked to create a syntax description file from a textual META description by a command of the form:

```
META filename subset_index
```

where filename.SYN is the name of a textual META syntax description. The subset_index (as described later) indicates those elements of the description to be eliminated to form a subset. The syntax description file to be created is given the name filename.SDF, where filename may be extended by digits from the subset_index. As an example the command:

```
META ADA
```

would create the syntax description file ADA.SDF from the textual description in ADA.SYN. The command:

```
META ADA $0
```

would create the syntax description file ADA0.SDF from the textual description in ADA.SYN while removing elements marked with subset 0.

3.2 Language Description

A META syntax definition is presented as a sequence of production rules, each defining some non-terminal in the language grammar. The first production rule in the description must define the non-terminal representing the language goal symbol. Appendix I specifies the format of a META description.

Each production rule may be either a concatenation or alternation definition. A concatenation is an ordered sequence of elements and represents a template to be laid into the program tree structure beneath a node which maps to the non-terminal being defined. Elements in a concatenation list may be conditional (options or repeaters). An option is an element enclosed in brackets ("[" and "]") and denotes an optional element. A repeater is an element enclosed in braces ("{" and "}") and denotes an element that may appear zero or more times. Options and repeaters may contain only a single element. The hide indicator ("!") may be used to mark conditional elements so they will not be automatically synthesized into the program tree. A concatenation list must contain at least one unconditional element to represent the production in a program tree.

An alternation is a list of alternatives, each a single unconditional element. The list is surrounded by angle brackets ("<" and ">") to distinguish it from a concatenation.

A syntactic element may be a non-terminal identifier, a terminal in the form of a literal string enclosed in quotes or a set construct. Each non-terminal must be defined exactly once in the syntax definition. Literal strings are typically used to represent reserved words and delimiters in the language grammar.

The set construct represents a compact means of expressing an alternation consisting of displayable ASCII characters. The set's alternatives may be single characters or pairs of characters which specify an inclusive range in the ASCII character set. META requires the set alternatives to be presented in ascending ASCII order. Sets are typically used in the specification of identifiers and numbers (as they are commonly called) whose individual character component values are determined during the synthesis process.

3.3 Format Controls

Program display formatting controls are embedded in the syntax description to allow the reconstruction of a textual display image from the abstract form of a program tree.

A space mark ("^") preceeding or following an element results in the placement of a corresponding space in the program tree image. A newline mark ("@") preceeding an element causes a new line to be generated in the program image followed by the proper number of tabs for the current level of textual indentation.

An indentation mark ("+") preceeding an element indicates that the current indentation level is to be increased and causes a new indented line to be generated. The entire program subtree beneath a node mapped to an indented element will be indented, after which the prior indentation level is restored.

Format controls for an element take effect only when that element is synthesized into the program tree. Format controls for conditional elements, therefore, must be placed carefully to insure that the appearance of a structure remains desirable with or without the presence of a conditional and whether or not it has been established.

3.4 Language Grammar Design

When creating a META description for a program language it may be convenient to examine existing definitions prepared in some other syntax definition language. Extended BNF definitions are particularly useful and require the least effort to translate.

Extended BNF notation, in general, allows a more complex form of expression than is available with META. Specifically, META disallows the use of complex expressions within options, repetitions and alternatives. This restriction generally requires an additional production rule to define a new non-terminal to replace a complex expression in an Extended BNF definition to create an equivalent META definition. As an example, the ADA reference manual defines an identifier list in a form of Extended BNF as:

```
identifier_list ::=
    identifier { , identifier }
```

META will not allow two elements ("," and identifier) within an option. The addition of a second production eliminates the problem resulting in the equivalent META definition:

```
id_list =
    identifier  { identifiers } ;

identifiers =
    "," identifier ;
```

Most of the problems encountered when translating from Extended BNF to META are of this type.

It is also important for the grammar designer to be aware of the impact particular decisions may have on the syntax-directed environment. The syntax-directed editor makes demands not normally required of a language grammar.

Achieving a desired display format and creating a natural and meaningful editing process for the programmer are human factors considerations which may require modifications to the language grammar. Conditional elements and alternations in a META grammar definition represent decision points or places requiring programmer attention during program editing. These situations should therefore be minimized where possible or placed where the decisions seem most natural.

If an alternation contains an element which is itself defined by an alternation, the user is required, when editing, to make two successive decisions. This can be reduced to one decision by including each alternative of the second definition as an alternative in the first. In this manner, the ADA reference manual definitions for primary and literal:

```
primary ::=
     literal | aggregate | name | allocator
   | function_call | type_conversion
   | qualified_expression | (expression)

literal ::=
     numeric_literal | enumeration_literal
   | character_string | null
```

were reduced to a single equivalent META definition:

```
primary = <
     decimal_number  name  nested_expression
     based_number  enum_lit  char_string
     func_call  "null"  aggregate  allocator
     type_conversion  qualified_expression >;
```

by including literal's alternatives within primary. The alternatives for numeric_literal were likewise moved to the definition of primary. These modifications were achieved at no cost since the non-terminals for literal and numeric_literal are referenced only once. Note also that the alternatives have been rearranged in an attempt to name the more frequently used ones first, the same order in which they will be displayed to the user for selection during the editing process.

The selection of non-terminal identifier names represents another design concern. Most names will appear at some time in the display of a program tree as a placeholder for an incomplete program fragment and to indicate syntactic type. Names chosen should therefore have some easily understood mnemonic value but should not be so long as to clutter the display screen.

Unfortunately, the limitations placed on language definitions by META may create complications in the editing process. The ADA reference manual definition for an identifier, for example, is:

```
identifier ::=
    letter  { [ underscore ]  letter_or_digit }
```

which eliminates the possibility of double or trailing underscores. An equivalent META definition removing the complex expression within the repetition would be:

```
identifier =
    'AZ|az'  { score_letter_digit } ;

score_letter_digit =
    [ "_" ! ]  '09|AZ|az' ;
```

This additional production requires an additional keystroke for each subsequent letter or digit. An alternative META definition, eliminating the additional production, is:

```
identifier =
    'AZ|az'  { '09|AZ|_|az' } ;
```

This provides a much smoother format for identifier entry at the expense of allowing illegal double or trailing underscores, errors which must be detected by the compiler. The added ease of identifier entry, a common occurence, was considered significant enough to warrant deferring detection of such trivial (and perhaps unnecessarily contrived) errors to the compiler phase. Fortunately, most design decisions do not require compromising the syntactic validity of program trees produced by a grammar.

3.5 Language Subsetting

Alternative and conditional elements in META may be marked with a subset index indicator which is a dollar-sign ("$") followed by a series of digits ("0" to "7"), each indicating a subset within which the element is too be restricted. The META preprocessor can be instructed, via the subset_index argument in the invocation command, to omit such elements when building a syntax description file from a META definition.

When all references to a non-terminal are removed by subset exclusion, its production rule is no longer required. All unreferenced rules are eliminated from the resulting syntax description file to conserve space.

VITA

Scott E. Ferguson was born on 31 May 1956 in East St. Louis, Illinois to Ed and Helen Ferguson. He attended high school at Belleville Township East in Belleville, Illinois as class valedictorian in 1974. In May of 1978 he graduated from the United States Air Force Academy where he earned a Bachelor of Science Degree in Computer Science. His first active duty Air Force assignment was with the 4501 Computer Services Squadron at Langley AFB, Virginia. He then entered the Air Force Institute of Technology at Wright Patterson AFB, Ohio in June of 1981 as a graduate student in computer science.

Captain Ferguson was married to Mary Stevens on June 7, 1978.

Permanent address: 1 Berkley Court
Fairview Heights, IL 62208

Unclassified

SECURITY CLASSIFICATION OF THIS PAGE (When Data Entered)

REPORT DOCUMENTATION PAGE

READ INSTRUCTIONS BEFORE COMPLETING FORM

1. REPORT NUMBER
AFIT/[illegible]/82D-1

2. GOVT ACCESSION NO.
AD-A124843

3. RECIPIENT'S CATALOG NUMBER

4. TITLE (and Subtitle)
A SYNTAX-DIRECTED
PROGRAMMING ENVIRONMENT
FOR THE ADA PROGRAMMING LANGUAGE

5. TYPE OF REPORT & PERIOD COVERED
MS Thesis

6. PERFORMING ORG. REPORT NUMBER

7. AUTHOR(s)
Scott E. Ferguson
Capt. USAF

8. CONTRACT OR GRANT NUMBER(s)

9. PERFORMING ORGANIZATION NAME AND ADDRESS
Air Force Institute of Technology (AFIT-EN)
Wright-Patterson Air Force Base, OH 45433

10. PROGRAM ELEMENT, PROJECT, TASK AREA & WORK UNIT NUMBERS

11. CONTROLLING OFFICE NAME AND ADDRESS
Air Force Avionics Laboratory (AFWAL/AAAF-2)
Wright-Patterson Air Force Base, OH 45434

12. REPORT DATE
December 1982

13. NUMBER OF PAGES
113

14. MONITORING AGENCY NAME & ADDRESS (if different from Controlling Office)

15. SECURITY CLASS. (of this report)
Unclassified

15a. DECLASSIFICATION/DOWNGRADING SCHEDULE

16. DISTRIBUTION STATEMENT (of this Report)

Approved for public realease; distribution unlimited.

17. DISTRIBUTION STATEMENT (of the abstract entered in Block 20, if different from Report)

18. SUPPLEMENTARY NOTES

Approved for public release: IAW AFR 190-17.
Lynn E. Wolaver
LYNN E. WOLAVER
Dean for Research and Professional Development
Air Force Institute of Technology (ATC)
Wright-Patterson AFB OH 45433

Approved for public release; IAW AFR 190-
Director of Information
14 FEB 1983

19. KEY WORDS (Continue on reverse side if necessary and identify by block number)

SYNTAX-DIRECTED
ADA
COMPUTER PROGRAMMING

20. ABSTRACT (Continue on reverse side if necessary and identify by block number)

This document describes the design and implementation of a programming support environment for the ADA language based on a syntax-directed editor and a program tree structure. Though the prototype compiler is limited to a small subset, the full ADA language is supported by the remainder of the environment. Most of [illegible] therefore capable of processing virtually any programming language. The prototype syntax-directed environment demonstrates the ability

DD FORM 1 JAN 73 1473 EDITION OF 1 NOV 65 IS OBSOLETE

Unclassified

SECURITY CLASSIFICATION OF THIS PAGE (When Data Entered) to reduce programmer idle time during development by eliminating parsing and lexical analysis in the compiler. The program tree structure also allows for the development of superior environment tools.

END

DTIC

8-86